Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Component-Based Software Engineering: a Quantitative Approach

Miguel Carlos Pacheco Afonso Goulão

Dissertação apresentada para a obtenção do Grau de Doutor em Informática pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Lisboa
(2008)

This dissertation was prepared under the supervision of
Professor Fernando Manuel Pereira da Costa Brito e Abreu,
of the Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa.

To my father, Manuel

[This page was intentionally left blank]

Acknowledgements

I would like to thank all those who have directly or indirectly helped me along the way.

To my supervisor, Fernando Brito e Abreu, for all his advices for so many years now, on research, teaching, life, and so many other things. He encouraged me to start my research career and has continuously supported me ever since, with his guidance, know-how, and endless suggestions for improvement on whatever we do, while providing me the freedom so that I could try new ideas and follow my own path.

To Pedro Guerreiro, Ana Moreira, João Araújo, Miguel Monteiro, and all the students of the Software Engineering research group. They have always been available to exchange thoughts and help me through this work, and were very pro-active in making me feel a member of the team since I joined the group. Fernando's students, including Aline Baroni, Sérgio Bryton, Eduardo Miranda, Vítor Gouveia, Filipa Silva, Ricardo Santos, and so many others, were particularly helpful in the last few years, with endless discussions on their work and mine, which have certainly helped me to mature as a researcher.

To all my other colleagues in the Informatics Department, for providing me with good companionship and a pleasant working environment. Luís Monteiro has always been supportive and has avoided overloading me with tasks which would distract me from my research. To Pedro Barahona, José Cardoso e Cunha, and Legatheaux Martins, for all their help as heads of the Informatics Department. I was extremely lucky to share the office with Armanda Rodrigues for several years. She has been a source of good mood and interesting discussions, in spite of the increasingly higher piles of papers and books she finds on my desk. Luís Russo was my personal trainer in how to postpone procrastination and other useful time management skills, during the final part of writing this dissertation. João Lourenço gave me the LaTeXtemplate for this dissertation. Adriano Lopes has been a good friend and a great help in understanding how things work in the University. José Pacheco has been the best example of an altruistic colleague since our undergraduate course and I am looking forward to seeing him finishing his own PhD dissertation soon.

To the secretariat of the Informatics Department of FCT, Filipa Reis, Paula Brás, Anabela Duarte, and Sandra Rainha for all their kind help and pro-activeness dealing with the bureaucracies during the dissertation. They were a great support, particularly

in the final stages of my doctoral preparation work.

To the organizers and mentors of the ECOOP'2003, SEDES'2004 and OOPSLA'2005 doctoral symposiums, for their challenging questions and useful feedback. To all the reviewers of the papers we have submitted during this work, for their comments and suggestions, which were very helpful in maturing this dissertation.

To all my friends, who made sure I had a social life, even when I thought I had no time for that. They were always there to celebrate the good moments, help me through the bad ones, and send me pesky short messages whenever my soccer team lost a match. They are too many to mention here, but nothing would make sense without them. So, let me just thank Nuno, Pedro and Ana, three brothers who sort of adopted me some thirty years ago, and have been around since then, the Pasta fans that have been with me for more than a decade, the Medeia fellows with whom I share countless evenings, the choir friends who made me a tenor feared all over the country, and my drama teachers, who just made me a better person.

Last, but not the least, I would like to thank my family. My father, Manuel, my lovely step-mother Fátima, and my brothers and sisters, Pedro, Jorge, Beatriz, Amélia and Maria, my sister-in-law Marta, my uncle Armando, my aunt Lourdes, my cousin Bates, and so many other cousins, who make sure I feel homesick when they go on vacation. And then, of course, there is Guida, my better half, the One, and her family, who I have learned to treasure in the last few years.

I would also like to acknowledge the following organizations for their financial support in the fulfillment of the research activities described in this dissertation: Departamento de Informática of the Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa (FCT/UNL); Centro de Informática e Tecnologias da Informação of the FCT/UNL; Fundação para a Ciência e Tecnologia through the STACOS project (POSI/CHS/48875/2002); The Experimental Software Engineering Network (ESERNET); Association Internationale pour les Technologies Objets (AITO); Association for Computing Machinery (ACM).

Summary

Background: Often, claims in Component-Based Development (CBD) are only supported by qualitative expert opinion, rather than by quantitative data. This contrasts with the normal practice in other sciences, where a sound experimental validation of claims is standard practice. Experimental Software Engineering (ESE) aims to bridge this gap. Unfortunately, it is common to find experimental validation efforts that are hard to replicate and compare, to build up the body of knowledge in CBD.
Objectives: In this dissertation our goals are (i) to contribute to evolution of ESE, in what concerns the replicability and comparability of experimental work, and (ii) to apply our proposals to CBD, thus contributing to its deeper and sounder understanding.
Techniques: We propose a process model for ESE, aligned with current experimental best practices, and combine this model with a measurement technique called Ontology-Driven Measurement (ODM). ODM is aimed at improving the state of practice in metrics definition and collection, by making metrics definitions formal and executable, without sacrificing their usability. ODM uses standard technologies that can be well adapted to current integrated development environments.
Results: Our contributions include the definition and preliminary validation of a process model for ESE and the proposal of ODM for supporting metrics definition and collection in the context of CBD. We use both the process model and ODM to perform a series experimental works in CBD, including the cross-validation of a component metrics set for JavaBeans, a case study on the influence of practitioners expertise in a sub-process of component development (component code inspections), and an observational study on reusability patterns of pluggable components (Eclipse plug-ins). These experimental works implied proposing, adapting, or selecting adequate ontologies, as well as the formal definition of metrics upon each of those ontologies.
Limitations: Although our experimental work covers a variety of component models and, orthogonally, both process and product, the plethora of opportunities for using our quantitative approach to CBD is far from exhausted.
Conclusions: The main contribution of this dissertation is the illustration, through practical examples, of how we can combine our experimental process model with ODM to support the experimental validation of claims in the context of CBD, in a repeatable and comparable way. In addition, the techniques proposed in this dissertation are generic and can be applied to other software development paradigms.

[This page was intentionally left blank]

Sumário

Enquadramento: As afirmações sobre o Desenvolvimento Baseado em Componentes (DBC) são, normalmente, suportadas pela opinião qualitativa de peritos, mas não por dados quantitativos. Este cenário, também observado na Ciência da Computação e em Engenharia de Software, contrasta com o de outras ciências, em que a validação experimental de propostas e teorias é prática corrente. A Engenharia de Software Experimental (ESE) procura mitigar este problema, mas frequentemente encontramos validações experimentais que são dificilmente comparáveis e replicáveis.
Objectivos: Nesta dissertação pretendemos (i) contribuir para a evolução da ESE por forma a favorecer a facilidade de comparação e replicação de validações experimentais, e (ii) aplicar as nossas propostas de evolução em problemas do DBC.
Técnicas: Propomos a combinação de um modelo de processo que procura reflectir as melhores práticas em ESE, com uma abordagem à medição denominada Medição Guiada pelas Ontologias (MGO). A MGO melhora as técnicas usuais de definição de métricas, formalizando e tornando executáveis essas definições, sem sacrificar a sua usabilidade. A MGO é baseada em tecnologias padrão, o que facilita a sua integração com ambientes de desenvolvimento de software modernos.
Resultados: Realizamos a uma validação do modelo de processo para ESE. A utilização da MGO no âmbito do processo para ESE facilita a definição e recolha de métricas para o DBC. Estas técnicas são usadas em vários casos de estudo, incluindo uma validação cruzada de métricas para JavaBeans, uma análise dos efeitos da capacidade individual no sucesso de equipas que realizam um sub-processo do DBC (inspecções sobre o código de componentes), e um estudo sobre padrões de reutilização numa arquitectura baseada em plug-ins (Eclipse). Cada estudo efectuado inclui a definição, adaptação, ou selecção de uma ontologia, bem como a definição formal de métricas.
Limitações: A versatilidade das nossas propostas é ilustrada pela variedade de modelos de componentes em que as aplicamos, bem como pela sua a aplicação sobre o processo e sobre o produto. Sobram inúmeras possibilidades de aplicação das técnicas propostas que contribuiriam para novas formas de validação das mesmas.
Conclusões: O principal contributo deste trabalho é a ilustração, através de exemplos concretos, de como o nosso modelo de processo para ESE pode ser combinado com a abordagem MGO para suportar uma validação experimental, replicável e comparável, de propostas feitas no contexto do DBC. A genericidade das nossas propostas também as torna adequadas a outros paradigmas de desenvolvimento de software.

[This page was intentionally left blank]

Contents

[This page was intentionally left blank]

List of Figures

[This page was intentionally left blank]

List of Tables

Chapter 1

Introduction

Contents

Background: *Experimental Software Engineering (ESE) is concerned with the design and execution of experimental work aimed at collecting empirical evidence that can be used to build up the Software Engineering body of knowledge. Component-Based Development (CBD) presents new challenges to the ESE community.*

Objectives: *Our goals are to introduce the notion of CBD, to motivate the usage of quantitative approaches to support it, and to outline the contents of this dissertation.*

Methods: *We characterize CBD, quantitative approaches to support it, and several shortcomings of those approaches. Our main contributions to mitigate those shortcomings are underpinned by a process model for ESE and an approach to software measurement called Ontology-Driven Measurement (ODM).*

Results: *This introduction motivates the discussion of the process model and ODM in the context of CBD, and outlines our contributions to advancing the state of the art of experimentation in the context of CBD.*

Limitations: *It is unfeasible to cover all the relevant component models and process models for CBD. While our proposals are generic and can be applied to several component models, as well as to the CBD process, their validation will use a set of selected examples, ranging from toy examples to real-world CBD projects.*

Conclusions: *Our work is aimed at conducting replicable experimental work and facilitating its meta-analysis by our peers, in the context of CBD, to facilitate advances in the CBD body of knowledge.*

1.1 Component-based development

The goal of **Component-based development** (CBD) is to achieve a rapid assembly of new software systems from existing software components [Bachman 00]. We can describe CBD's main activities through a software process model. A **software process** is a set of activities, and associated results, which produce a software product. A **software process model** is an abstract representation of a software process [Sommerville 06]. The fundamental activities, common to most software processes, include software specification, design, implementation, validation and evolution.

In the context of CBD, where component-based systems are, for the most part, built from existing third-party components, one should consider not only (i) the software process of **developing a component-based system**, but also (ii) the software process of **developing software components**. The components to be integrated in a new component-based system may be already developed, tested, and used in other projects, when the system development process begins. In that sense, they are independent from the systems they may have been integrated in. A good example of such components are the so called Commercial Of The Shelf components (COTS). COTS components may range from fine (e.g. a calendar component) to coarse-grained (e.g. a SQL database manager component). This separation between component users and component developers, implies a third process for (iii) **finding and evaluating components**.

A discussion on these three parallel processes and their combination in a family of process models for CBD can be found in [Crnkovic 06]. The heterogeneity that may result from following different approaches to CBD, such as Architecture-driven CBD, product line development, and COTS-based development, leads to specific process models for each of these approaches. All the process models in this family include processes (i) through (iii) as intrinsically separate sub-processes of the CBD process that can be carried out by independent organizations. As such, they lead to three different practitioner profiles, each with his own concerns.

Consider a component-based system developer:

- While defining the requirements for the component-based system and the subsequent design, system developers are concerned with the availability of existing components that may be reused. They need to identify the most suitable components for their needs and integrate those components, but often have neither access to the component implementation details nor control on the evolution of those components.

- System developers may not find a perfect match to their requirements. This may lead to the adaptation of the component (e.g. by using a component wrapper), or even, inversely, to a change in the requirements, to conform with the provided features.

- System developers have to test the component assemblies, to check whether the combination of the chosen components conforms to the requirements or not. Current component models are not well suited to allow a safe prediction of the final component-based system properties based on the properties of the individual components.

- When a new version of a component replaces a previous one, system developers may have to readapt their assembly to support the usage of the new version of the component.

From the point of view of a component developer, the following concerns can be highlighted:

- Often, component developers build their components without fully knowing where and how those components will be reused in the future. They can test their components in isolation, but they can not anticipate all the possible interactions their components may have with other components they are integrated with.

- While evolving components, there is a risk of breaking the compatibility with the component-based systems where the previous versions of those components are used. This risk stems from explicit and implicit dependencies that may exist on the previous version of the component. While the former are supported by current component technologies, e.g. through a standard interfaces description language, the latter are not. An example of implicit dependencies for which current component technologies provide inadequate support is the representation of non-functional properties. The lack of an explicit standard representation of such properties may lead to undeclared dependencies on them, and those dependencies may be broken in the new version of the software component.

Finally, those engaged in finding and selecting components deal with a different set of problems:

- Using the requirements provided by system developers, they must find the components that best match the requirements. This involves not only finding candidate components, but also being able to compare and rank them, as it is often the case where no perfect match can be found.

- Component selection involves testing the components in isolation and integrated with other components. These tests should cover functional and non-functional properties.

- Components identified as good reuse candidates should be stored along with meta-information in a component repository, to facilitate their reuse in other contexts.

- Finding and evaluating components for reuse can be performed as a service of a component repository provider, to facilitate the selection of software components for reuse, thus leading to economies of scale with respect to component assessment. On the other hand, this clear separation between component users and component selectors is not without its costs: for instance, testing the components without access to the assemblies they will be integrated in becomes more complex.

CBD fosters reductions on the development costs and time to market, and improvg the developed system's overall quality [Szyperski 02]. These improvements stem from the reuse of software components. The extra effort required for selecting, evaluating, adapting, and integrating components is mitigated by avoiding the much larger effort that would be required to develop the functionality of such components from scratch. As components get to be reused in several systems, with different requirements, they tend to become well tested and robust pieces of software, thus contributing to the increase of the final system's quality.

The challenge is that existing components have to be evaluated, selected and, often, adapted to suit the particular needs of the software system being assembled, frequently without access to the component's source code. This constrains the extent to which we can assess such components, and differentiates component assessment activities from those available for white-box software reuse. As there is no generally accepted standard for assessing components, the evaluation of software components is often carried out in an ad-hoc fashion. However, if evaluations are not independently and consistently replicable, their success depends highly on the assessor's expertise.

1.2 Current state of the art

The bulk of research in CBD has been devoted to the functionality and composability of software components. As a young discipline, Component-Based Software Engineering (CBSE), which is the branch of Software Engineering dedicated to CBD, is still focused on technology issues, such as modeling, system specifications and design, and implementation. The area of assessment for CBD remains unexplored. For instance, there is no widely accepted quality model suited for CBD assessment [Simão 03], although there have been attempts to adapt the ISO9126 quality model [ISO9126 01] to CBD, such as the model proposed by Bertoa and Vallecillo [Bertoa 02], which uses a subset of ISO's quality attributes. Some of those attributes are redefined to better reflect the specificity of CBD. Washizaki *et al.* proposed a quality model for components reusability [Washizaki 03], and Bertoa *et al.* focused on the usability of COTS components [Bertoa 06].

The ability to predict the system's properties from the properties of reused components is a growing concern for the research community [Crnkovic 04]. Some proposals

aim at developing prediction-enabled component specifications to facilitate automated prediction of properties [Wallnau 03, Larsson 04]. They focus on the analysis of run-time quality attributes, but the effectiveness and the feasibility of their method require further validation.

A complementary research area, where we have been conducting our research, is static analysis of quality attributes, such as reusability or maintainability, using software metrics [Goulão 05c, Goulão 05a, Goulão 05b]. Several authors contributed with proposals for the evaluation of component interfaces and their dependencies [Boxall 04, Gill 04, Washizaki 03], with a particular concern on their reusability. Others use metrics to assess component packing density (the density of a given constituent element, such as operations within the component) as an indirect measure of the component's internal complexity [Narasimhan 04]. Bertoa *et al.* defined and validated a set of component usability measures [Bertoa 06]. Their metrics set combines metrics related to the complexity of the design with others related to the documentation of the components. All of these proposals take a component-centric approach, meaning they assess components in isolation.

Wallnau and Stafford argue that it is more effective to perform the evaluation on assemblies, rather than on individual components [Wallnau 02]. Component-based systems' developers are concerned with selecting the components that maximize the overall system quality. In this assembly-focused view, individual component assessment may be performed as part of the component assembly evaluation, but the focus is on selecting the overall best solution with respect to the quality attributes one is trying to maximize. Examples of metrics following this view can be found in [Narasimhan 04, Hoek 03].

The existing quality models and metrics include informal specifications. Even when the metrics are specified through mathematical formulas, the elements in those formulas are usually expressed in natural language. This creates ambiguity, if there are several plausible interpretations for such definitions. Ambiguity is an obstacle for independent validation efforts. Such experimental replication effort is essential to a sound validation of quality models and metrics proposals. Furthermore, tool support for metrics collection is usually unavailable, as their proponents either did not build collection tools or do not provide access to them. With few exceptions, such as [Washizaki 03, Bertoa 06], most proposals went through scarce validation, if any.

In the absence of a systematic and feasible approach to quantitative evaluation, an alternative is to adhere to a qualitative solution, i.e. the opinion of an expert. Based on their experience and on a subjective quality model, experts make informal judgments that are hard to replicate. Moreover, experts may not be available to perform the assessment. In such an event, practitioners often perform relatively blind choices, with respect to the quality of components. With the growing reuse of components in software construction, such choices are a threat to the success of CBD projects.

Another unexplored research area is that of the CBD process. Many principles of CBD influence the development and maintenance process and require considerable modifications to more traditional development processes. Currently, there is no widely accepted CBD process model. The ability to perform assessments at a CBD process level should provide the research and development communities with appropriate approaches to facilitate the comparison between alternative software processes.

In summary, we identify the following problems:

- there is no widely accepted quality model for CBD;
- existing quality models were not independently validated;
- most of the existing metrics are unrelated to a quality model, and some of them have unclear specific measurement goals;
- most of the existing metrics lack sufficient validation;
- existing metrics definitions use an inadequate formality level;
- most of the metrics for CBD are designed to assess components in isolation, but components should also be assessed in the context of the system in which they are to be integrated;
- current assessment practices rely mostly on subjective expert's opinions;
- CBD process definition and assessment have not been studied in detail.

1.3 Contributions of this dissertation

A way of mitigating the problems identified in the previous section is to evolve the integrated development environments (IDEs) so that they include functionalities to facilitate quantitative assessment in CBD. An analogy can be made to the automated refactoring functionalities of some IDEs. Without the inclusion of these refactoring functionalities, their usage would have a smaller impact on the current state of practice in software development. The validation of metrics to assess components and assemblies in the context of well-defined measurement goals is a pre-requisite for such IDE evolution. Practitioners should be able to know when and how to use quantitative information to support their work. The integration of such support in IDEs is essential, if a widespread adoption of a quantitative approach to CBD is sought.

In this dissertation, we demonstrate the feasibility of a formal approach to CBD assessment that combines rigor with simplicity, is replicable, and can be integrated with current development environments, thus providing automated advice to practitioners involved in CBD assessment. This approach, called Ontology-Driven Measurement (ODM), is an evolution from the MetaModel Driven Measurement (M2DM) approach,

originally proposed in [Abreu 01b, Abreu 01a] for the evaluation of object-oriented designs using software metrics.

We extend the concept of M2DM to models which may not necessarily be metamodels. This evolution is required so that we can define metrics at different meta-levels, according to the requirements of each experimental work. In some situations, we use a metamodel, while in others we use a model. In any case, our metrics will be defined using an ontology of the domain upon which we want to perform the measurements. The nature of this domain is also an evolution from M2DM's original proposal: while M2DM was created to allow defining and computing metrics in object-oriented designs, in this dissertation we will use ODM to define and compute metrics both at the process and at the product level, in the context of CBD.

Our main contributions include:

- Ontologies (metamodels and models) construction and extension.
- The formalization of metrics for CBD, using several underlying ontologies, to explore the expressiveness of the ODM approach.
- Experimental validation of our proposals, by using a common software experimentation process model, also proposed in this dissertation.
- Development of prototypical tool support for the experimental activities described in this dissertation.

We regard **ontology construction** as a first step towards assessment in CBD. Expressing the CBD concepts relevant to a particular measurement initiative through a ontology helps eliciting such concepts and their relations. In some situations, an existing ontology can be readily adopted (e.g. when UML 2.0 components are assessed, we can use the UML 2.0 metamodel [OMG 07]). In others, we have to either extend an existing ontology, or to create a new one, in order to reach adequate expressiveness for our assessment tasks. The lack of a generic, widely accepted, component model hampers the adoption of a generic ontology for CBD.

While preparing this dissertation, we proposed several metamodels and extensions. In [Goulão 03], we created a UML profile representing Acme [Garlan 00b] components in UML 2.0. We created this profile to assess the suitability of UML 2.0 as a component Architecture Description Language (ADL), with respect to the structural view of component-based architectures. We chose to create a profile for the Acme language, because Acme was originally designed to capture the main structural concepts of ADLs. We concluded that UML was indeed suitable for expressing the structural views of although it has less syntactic sugar than Acme, particularly for expressing synchronous communication between components. While defining metrics upon the standard Corba Components Metamodel (CCM) [OMG 02a] we identified some limitations in that metamodel, concerning the specification of component assemblies.

In [Goulão 05a], we proposed an extension of the CCM to support the representation of the instantiation of component assemblies. We will discuss it in chapter 5. The experimental work carried out in chapter 7 lead to the definition of a metamodel for representing Eclipse plug-ins. We also define a process model for representing part of the component development process, in chapter 6 [Goulão 06], in order to support measurement at the process level.

The quantitative assessment of components and component assemblies requires a rigorous approach. Different assessors evaluating the same component or component assembly in different locations must be able to replicate the assessment conditions and get the same results. This requirement of replicability is generic to scientific experimentation and is earning a growing attention in industry and academia, although its fulfillment remains a challenge (see, for instance, [Jedlitschka 04]). The driver behind the replicability requirement is the ability to offer evidence on the effects of using a particular technology, or technique, rather than providing a set of toy examples and unconvincing claims. This concern crosscuts the whole dissertation and is made explicit through the process model for conducting experimental work, in chapter 3 (originally proposed in [Goulão 07a]).

Using an ontology-based approach to metrics collection experiments, at a process and product level, reduces the subjectivity in the experiment design, thus facilitating its replication. Moreover, expressing metrics definitions formally upon an ontology removes ambiguity from the definitions, and provides an executable way of collecting the metrics. Furthermore, the technique used in the metrics definition can be extended for defining heuristics that help assessing the metrics results, thus providing a stronger integration with the underlying measurement goals.

As a proof of concept with respect to the expressiveness of the metrics specification technique, we formalize the definition of metrics available in the literature as well as propose new metrics for CBD. The formalized metrics include not only metrics for individual components (both in isolation and within a specific component assembly), but also for component assemblies. This formalization (as well as that of heuristics) is carried out upon several component models.

The usefulness of a quantitative approach to support assessment in the scope of CBD is carried out through the experimental work presented in this dissertation. This work includes:

- A case study for the cross validation of a metrics set proposed by Washizaki *et al.* [Washizaki 03], in chapter 4 [Goulão 05c].

- A quasi-experiment on the influence of practitioners' expertise in one of the process activities carried out during software component development, in chapter 6 [Goulão 06].

- An observational study on the reusability of software components in a large

repository.

For each of these experimental validations, the set of research goals, the underlying ontology, the metrics definitions and collection, and their interpretation with respect to the research goals are discussed.

All the experimental work presented in this dissertation follows the experimental process model defined in chapter 3. While the experimental work presented in this dissertation provides a set of anecdotal examples of the usage of the experimental process model, we also include a case study in chapter 3 to evaluate the process model itself. Naturally, this case study was also conducted using the process model.

The experimental work presented in this dissertation required developing **tool support** to define and collect software metrics. The architecture of such tool support will also be discussed, with a focus on how it can be integrated with a modern IDE for providing automated support for assessment in CBD. The integration of assessment tools with common IDEs is essential to foster its usage and has influenced our choices of the formalization technique, as discussed in the next section.

1.4 The approach

Our overall approach to adopting Experimental Software Engineering practices in the context of component-based development can be viewed as an instantiation of the process model that we will describe in detail in chapter 3.

While studying the problem of selecting, or creating, a quality model for CBD (see, for instance, [Goulão 02a, Goulão 02b]), we concluded that defining fine-grained quality models aimed at specific niches of CBD is more feasible than aiming for a general CBD quality model. The diversity of issues that such a generic quality model involves would lead to a quality model too complex to be easily grasped by practitioners and therefore useless. This option for quality models aimed at specific niches is supported by the analysis of work of [Bertoa 02, Bertoa 06], where we observed how an evolution from a generic quality model to a specific one facilitated the feasibility of their validation.

Although we do not propose new quality models in this dissertation, we present quality concerns that underpin the goals of each of the experimental works described throughout this dissertation. This is followed by a Goal-Question-Metric approach [Basili 94], to determine which metrics should be collected to assess components, assemblies, or some process aspect.

Figure 1.1 outlines the metrics collection process. We use a domain ontology to express the basic concepts from which we want to extract relevant properties. Then, we populate the ontology with an instantiation that represents the target to be assessed. For instance, in the most frequent case in this dissertation, the ontology is a metamodel. Therefore, the instantiation is a graph where the nodes are meta-objects and the edges

are meta-links. We use OCL expressions which define the metrics to be collected to traverse the graph and compute the metrics. Heuristics definitions may also be defined in OCL, at this stage. Finally, we analyze the results of the OCL expressions - both metrics and heuristics.

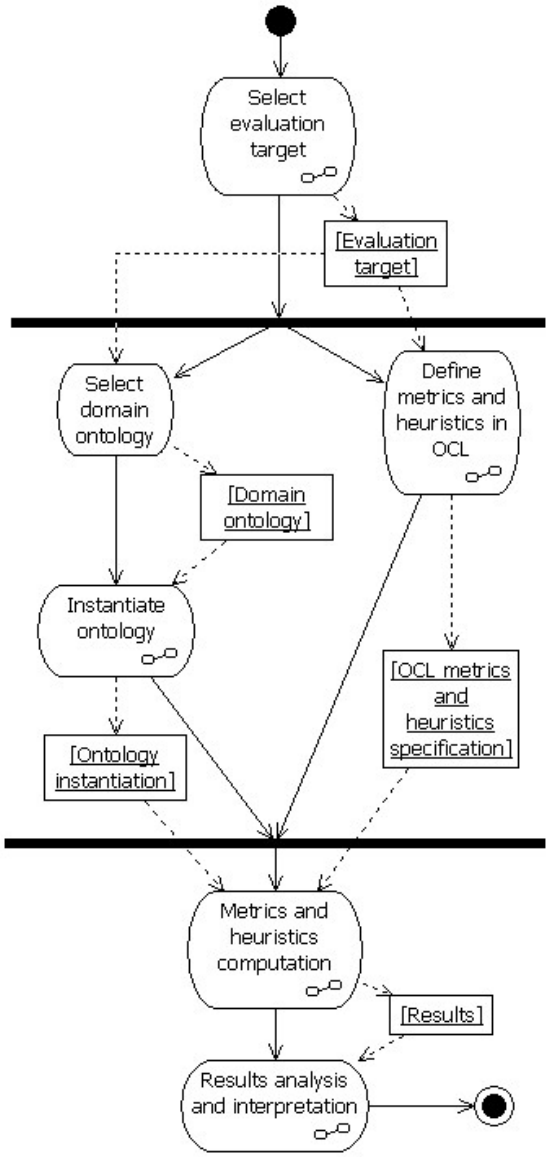


Figure 1.1: Metrics collection process

The OCL provides the required formality without sacrificing understandability, since it was conceived with usability in mind for UML practitioners.

The approach itself is generic: we can use it for assessing products and processes. For products, we will use ODM for assessing both individual components and component assemblies in chapters 4, 5, and 7. These assessments are carried out using a variety of component models, including JavaBeans - chapter 4 -, UML 2.0 components, the CORBA Component Model (CCM) - chapter 5 -, and Eclipse plugins - chapter 7.

The usage of ODM in process assessment is illustrated in the the quasi-experiment

described in chapter 6.

The approach is also flexible: adding a new metric, requires defining a new OCL expression that specifies how the metric should be computed. Heuristics may also be defined using the same technique, typically through the specification of OCL predicates that check for metrics values outside their expected range [Goulão 04a, Goulão 05c].

The approach is open in the sense that the metrics are defined using standard OCL clauses, and it only requires a common UML tool with OCL support, upon which we can load a model (the domain ontology) and populate it with the appropriate instances. We are currently using the USE tool [1] [Richters 01] for this purpose, but this kind of computational support is becoming increasingly available in several UML tools, as they become "OCL-enabled". Several of those tools support the UML 2.0 component metamodel, either internally (on their data dictionary) or at least through their external interface (e.g. by providing an import/export feature using UML 2.0-compliant XMI). By basing our approach in a *de facto* standard such as UML 2.0, combined with OCL, we enable a smooth integration of our proposals into current and future IDEs that support UML 2.0 and OCL.

For other component metamodels, we will still have to go on developing instances generators, or, in alternative, to use UML profiles, such as the one for EJB, so that we can specify and collect the metrics on top of the UML 2.0 metamodel and the used UML profile.

The evaluation of our proposals is carried out in four different ways:

- The work described in chapters 4 through 7 included setting up an assessment infrastructure and exercising it with different underlying ontologies and metrics for different purposes. It covers covering different aspects of component interfaces and interaction mechanisms, at the product level. It also covers different parts of the software process, namely code inspections and software maintenance and evolution. By doing so, we are able to assess the flexibility of our approach.

- The peer review of our work in the context of international scientific forums provided us with an external assessment of the soundness of our proposals. The vast majority of the contents of chapters 2 through 6, as well as appendix B has been published in peer reviewed journals, conferences or workshops.

- The primary source of validation of our proposals consists on the experimental work in chapter 6 [Goulão 06], and the observational studies described in chapter 7. A case study for the validation of the proposed experimental approach is also presented, in chapter 3.

- We also contribute to the external validation of proposals by other authors, as in the cross-validation case study presented in chapter 4 [Goulão 04a, Goulão 05c].

[1] http://www.db.informatik.uni-bremen.de/projects/USE/

1.5 Dissertation outline

Figure 1.2 outlines the contents of this dissertation. The dissertation is organized into four parts. In the first part we provide an introduction and some background, in order create an adequate framework for the discussion on the usage of experimental techniques to support CBD, both in what concerns the development of components for reuse and the development of component-based software. It includes chapters 1 and 2. The second part presents our research contributions and is focused mainly in ODM. It includes chapters 4 and 5. As we can observe in figure 1.2, chapter 3 merges characteristics of a background chapter with those of a contributions chapter. The reason for this hybrid nature is that we chose to provide the background on Experimental Software Engineering through the proposal of a process model for it. So, we can regard chapter 3 as a transition chapter from the first part of this dissertation to the second one. The third part of this dissertation is concerned with the experimental validation of our claims, and includes chapters 6, and 7. The experimental reports presented in these chapters can be safely visited in any order the reader might prefer. Finally, the fourth part of this dissertation contains chapter 8, with the dissertation's conclusions and our view on possible extensions to our work.

Chapter 1 introduces the theme of this dissertation, outlines the current state of the art in CBD, its main shortcomings, and how our work helps overcoming them. We briefly discuss the research approach, both with respect to the proposed solution and to the validation strategy for our work.

Chapter 2 provides a background on CBD, both with respect to the definition of software components and component models, and to the discussion of the changes that underpin CBD, when compared to other software development approaches. The chapter also includes a discussion on the state of the art of component-based software assessment and its current challenges to the community.

Experimental Software Engineering is at the heart of this dissertation. Chapter 3 is devoted to it, in particular to the modeling of the experimental process. It contributes a process model for conducting experimentation in Software Engineering. The model is generic to Software Engineering. We use it as a common process framework for all the experimental work described in the remaining of the dissertation. So, although the process model is generic, we will present several instantiations of it with examples of Experimental Software Engineering dedicated to CBD.

In chapter 4 we introduce Ontology-Driven Measurement (M2DM) as the fundamental approach used throughout this dissertation to deal with the technical challenges of the software measurement needs raised by the experimental process described in chapter 3. In chapter 4 we provide the tools to support the data collection part of such process, through the definition of metrics in OCL upon appropriate ontologies. We illustrate ODM with the cross-validation of a component metrics set for

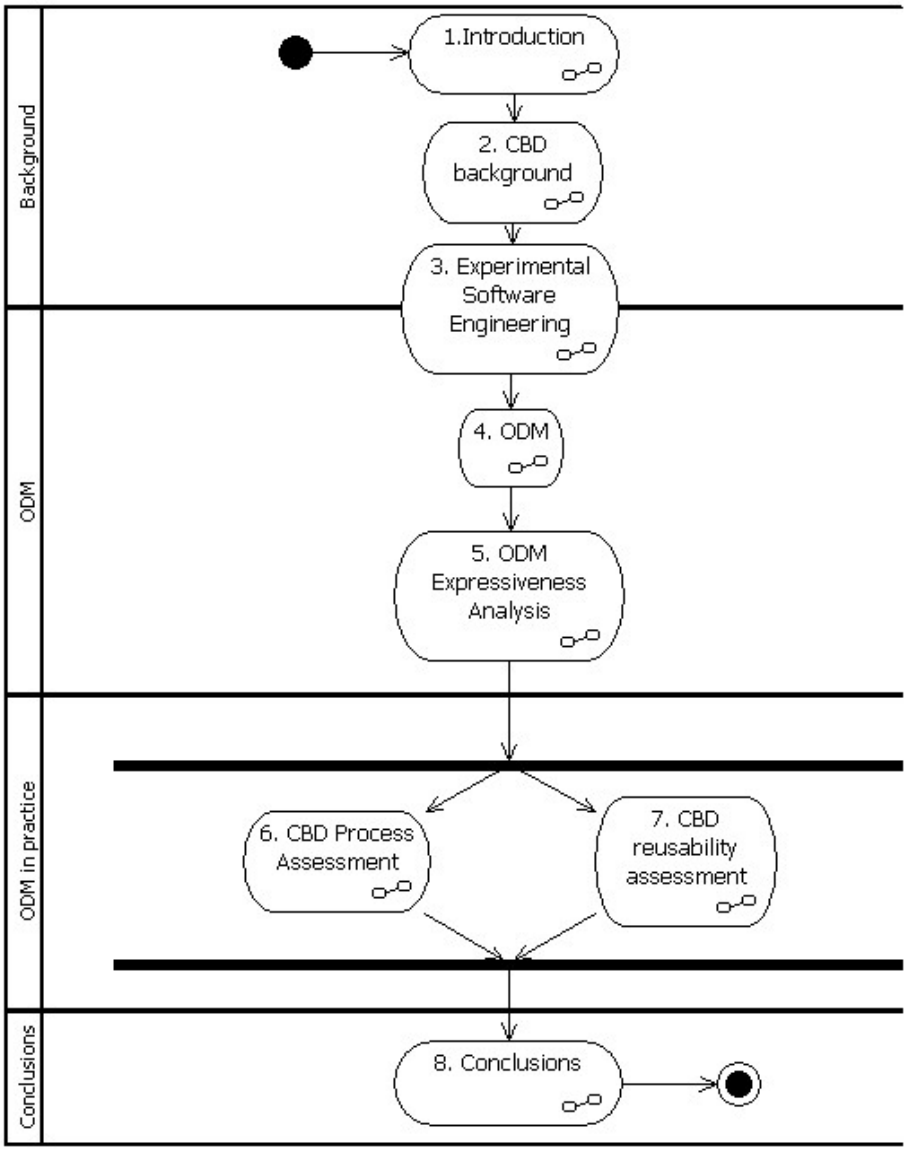


Figure 1.2: Dissertation outline

indirectly measuring the reusability of JavaBeans.

Chapter 5 discusses the expressiveness of ODM. It starts by introducing a set of examples specified using UML 2.0, and CCM. Then, it presents a metrics set covering metrics for components and component assemblies. The metrics set addresses different aspects, ranging from interface complexity to the effective reuse level of the components within a component assembly. We formalize the metrics for each of the component models, thus supporting the discussion on the expressiveness of the approach.

Chapter 6 presents a controlled experiment on the component development process. It focuses on code inspections conducted during component development, and on the effect of the level of expertise of inspectors in the outcome of the inspections. It illustrates how ODM may also be used to support the process assessment.

Chapter 7 presents an observational study on reusability patterns in open source component software. The sample used in this study is built from Eclipse plugins,

which are a particular kind of software components. We analyze reusability based on the public information provided by the Eclipse plugins, through their manifest files. Again, ODM is used to specify and compute metrics that support reusability assessment.

Chapter 8 presents the overall conclusions of our work and outlines future research streams that emerge from it.

Although not represented in figure 1.2, the dissertation includes three appendixes that complement the information in the main text.

Appendix A is dedicated to a review of existing component models. This review details the information provided in chapter 2, concerning those models. While in chapter 2 we adopt a set of criteria for conducting our review in a systematic way, and then summarize our observations, in appendix A we provide more details on each of the reviewed component models, following the criteria defined in chapter 2.

Appendix B is dedicated to a mapping between Acme and UML 2.0., referred to while describing the contributions of this dissertation. We created this mapping as an expressiveness assessment of the UML 2.0 notation, when compared with the core features of ADLs. This allowed identifying the strengths and shortcomings of the UML metamodel as an ADL. Furthermore, this mapping is a facilitator for quantitative experiments using components specified in other ADLs. The rationale is that Acme itself is considered as an interchange language for specifications in several ADLs. By providing a bridge from Acme to UML 2.0, we have created an indirect mapping from those ADLs that can potentially be used to express components and component assemblies upon a UML 2.0 profile, and then use ODM, with our Acme profile as the ontology.

Finally, in appendix C we discuss the architecture for the tool support created during this dissertation. This tool support is used in our experiments, to implement the ODM approach and subsequent statistical data analysis.

Chapter 2

Component-Based Software Engineering

Contents

***Background**: Before addressing the usage of an Experimental Software Engineering (ESE) approach to Component-Based Development (CBD), it is useful to define some basic concepts in CBD.*

***Objectives**: We present an overview of software components definitions, models, and technologies, as well as a discussion on the main peculiarities of the CBD process.*

***Methods**: We provide a narrative overview of the basic concepts of CBD, and present two systematic discussions on component models and metrics proposals for CBD.*

***Results**: We characterize current CBD technologies and identify shortcomings in current quantitative approaches to assessment in CBD, including problems with the context for those metrics, definition formalism, and insufficient validation of proposals.*

***Limitations**: The plethora of component models and metrics proposals for CBD would make unfeasible the inclusion of all of them, so we discuss a representative set of each.*

***Conclusions**: The discussion on CBD and component models provides the background, while the identification of shortcomings in the quantitative assessment in CBD motivates our proposals, in the remainder of the dissertation.*

2.1 Component-based development

As software becomes ubiquitous and increasingly sophisticated, there is a demand for improved software development processes and techniques that allow practitioners to tame software's growing complexity, while reducing development costs and time to market. Software reuse has been regarded as one of the keys to face this challenge. Reuse is a process for creating software systems from existing software pieces rather than developing them from scratch [Krueger 92].

The concept of reuse in the software development context was introduced by McIlroy almost forty years ago. He proposed the creation of a software component industry that would offer families of routines for any given job [McIlroy 69].

While opportunistic reuse (by cut and paste of code from old systems to new ones) has been used by individuals and small teams, it does not scale up well to larger organizations and complex software systems [Schmidt 99].

As proposed by McIlroy, early approaches to reuse were mostly based on the inclusion of function libraries (e.g. the C Standard Library [Plauger 91]). Later, with the shift to object oriented programming, class libraries became a common reuse asset (e.g. the C++ Standard Library [Josuttis 99]).

Both cut and paste coding and the usage of library functions, or classes, can be considered fine-grained approaches to reuse. Cut and paste coding is a dangerous form of reuse, in that it leads to the proliferation of code clones throughout the source code. If a bug is found in a portion of code which has been reused through cut and paste, or a requirement that lead to its creation changes, then producing the required modifications in that piece of code is expensive, as it is replicated in several different clones.

The reuse of library functions and classes has achieved a large success in the software development community. Widely used programming languages, such as Java, rely on a fairly small core, along with large libraries of classes aimed at simplifying development by providing commonly used abstractions to the programmer, such as support for handling collections, and other frequently used data structures and algorithms.

A shortcoming of fine-grained reuse is that it relies on fairly low-level units of abstraction. Function libraries and class libraries are not adequate for supporting coarser-grained reuse. Frequently, the elements of these libraries have to be combined so that we can obtain the desired functionality. So, we have a mismatch between the abstraction level of the reuse assets and that of the reuse requirements. This has lead to the adoption, by the software industry, of other, more sophisticated forms of reuse, such as the reuse of design knowledge, through design patterns [Gamma 95] and the reuse of component frameworks for specific domains (e.g. graphics components, for building graphical user interfaces).

A **design pattern** is a general design solution for a problem that occurs frequently in software development. This design solution is generic, rather than instantiated into a particular problem, so it can be thought of as a template of a design solution. The rationale is that these patterns can speed up the development process by providing a well-known solution to a common problem, so that developers can avoid "reinventing the wheel". Furthermore, patterns improve the readability of the design, at least in the perspective of developers who are familiar with those patterns, and therefore able to recognize them when analyzing a software design. Pattern catalogs, such as [Gamma 95] provide a common terminology for software designers, which includes not only a description of the design issues the pattern covers, but also how the pattern should be implemented, as well as when it should and should not be used. A shortcoming in patterns is that, for the sake of their reusability, they may provide a solution which is too generic and complex for a specific problem, leading to an implementation that is less efficient and more expensive to develop than a fine tuned alternative. This occurs when developers overuse patterns [Kerievsky 05]. This shortcoming is essentially similar to that of speculative generality, a bad smell in code that occurs when the code implements functionalities to handle unasked for requirements [Beck 99].

Meyer and Arnout have argued that the need to implement the design patterns is a major shortcoming, as developers have to re-implement the design patterns whenever they want to reuse them, thus making pattern reuse a "concept" reuse, rather than full reuse [Meyer 06]. They set out to implement in Eiffel [Meyer 92b] a well known patterns catalog [Gamma 95] as a set of reusable components, one for each pattern. Meyer and Arnout claim that, out of 23 patterns, this process, which they named "***componentization***", was fully successful with only 11 of them. 4 patterns were only partially *"componentized"*. They were not able to *"componentize"* the remaining 8 patterns, although they were able to provide some automated support for their integration in a component library (e.g., through the automatic production of component skeletons) for 6 of those 8 patterns.

In spite of reusability being considered an important quality attribute in software [ISO9126 01], several authors still argue that reuse has not yet fulfilled its true potential, although it remains a promising approach to the efficient development of high quality software [Heineman 01, Crnkovic 02, Szyperski 02, Inoue 05]. Difficulties to effective reuse can include both technical and non-technical aspects. It takes skill, expertise, and domain knowledge, to identify the best opportunities for developing or integrating reusable assets. The *"not invented here"* syndrome [1] is also a common obstacle to reuse. Difficulties to effective reuse include locating the appropriate reuse assets, adapting them to one's particular needs, and coping with those assets' evolution (which the reuser does not control).

A recent systematic review, by Mohagheghi and Conradi, on industrial studies con-

[1] This syndrome denotes the unwillingness to adopt an idea, or product, because it was not created in-house [Katz 82].

cerning the benefits of software reuse [Mohagheghi 07] shows that although there are several cost-benefit models for reuse, there is little empirical evidence from industry concerning the economic benefits due to reuse. Several of the studies reviewed supported the economic benefits claim, but their results were not statistically significant. However, improvements concerning reducing problem density and rework effort through reuse were reported consistently and significantly in several of the reviewed studies.

In our opinion, this contrast in the strength of evidences on the benefits of reuse may result, at least partially, from the difficulties inherent to each sort of analysis. Verifying the economic benefits due to reuse implies a data collection effort which is harder to enact, when compared to studying the effect of reuse in problem density. It is usually difficult to obtain reliable information concerning the effort of development teams, while a problem reporting system may automate the data collection required for assessing problem density. Effort information is also a problem when assessing rework effort, but perhaps the slightly more self-contained nature of this effort helps explaining the increased success of experimental work on rework effort, when compared to the one concerned with the economic benefits of reuse.

Mohagheghi and Conradi also profiled the reusable assets (e.g. functions, modules, and components) in software development. The fine-grained reuse of modules, or functions, favors small assets with as little external dependencies as possible. For coarser-grained reuse, such as the reuse of software components, the most noticeable driving factor for reuse is complexity. Encapsulating complex design in reusable assets helps reusers to benefit from the expertise of the asset's producers.

CBD is an approach to software development that relies on the reuse of existing software components to reduce the development costs and cycle, while increasing the final product's quality. A case study conducted by a component broker, in cooperation with software component producers on the return on investment of using COTS reported that the costs of acquiring such components were about 1/50 of the ones of developing their required functionalities from scratch [Brooke 02], although, as we have seen earlier, such benefits are yet to be confirmed through independent empirical validation.

CBD involves several risks both for component producers and users. From the point of view of component producers, the time and effort required to build reusable components, is likely to be higher, when compared to the one needed to build a solution specific piece of software. There is also a trade-off between usability and reusability, as generic, scalable, adaptable components tend to be more complex and resource consuming than their specific counterparts. Components may have unclear or ambiguous requirements: they are typically reused in different systems that may have conflicting requirements, and are often built independently from the systems they are later reused in, which adds uncertainty in what concerns component requirements

specification. Furthermore, components and the systems where they are integrated in have separate life cycles. This is a concern for component users and producers. The former may have no control on the evolution of the components they are reusing, while the latter have no control on the evolution of the software their components are reused in. On one hand, this separation of life cycles may lead to ripple effects in maintenance, in the event of changes either on the component or on the application side. In other words, maintenance actions in one of the sides may induce the need for maintenance actions on the other side, thus leading to increased maintenance costs. On the other hand, the separation of life cycles may also lead to economies of scale, as the same component gets reused in several different contexts. Components are often distributed as black-boxes. Therefore, component users have limited knowledge with respect to the component properties, both functional and non-functional, which may lead to behavior that do not match the expectations.

The area of Software Engineering that deals with CBD is called Component-Based Software Engineering (CBSE). The Software Engineering Institute (SEI) defines CBSE as being *"concerned with the rapid assembly of systems from components where components and frameworks have certified properties, and these certified properties provide the basis for predicting the properties of systems built from components"* [Bachman 00]. Their vision on CBSE, encompasses both the developer's ability to manage quality concerns and the market pressures (rapid assembly of systems from components) that are key motivations for CBD, such as a short time to market, combined with lower costs. In particular, SEI's vision encompasses the ability to use the individual component properties to predict the component assembly properties. As we will see while discussing current component models both from industry and academia, the quest for predictable assembly properties is an active topic of research, as current component technologies provide little or no support for it.

2.2 Software components

Industries such as the automobile, building, computer hardware or consumer electronics deeply rely upon the availability of application-domain specific components. Those components are often standardized, therefore allowing their availability from multiple sources. In those industries, developing a project is often an exercise of components selection (from catalogs of available ones) and composition (gluing the components together). This industry metaphor for developing a project by selecting and composing components is often used when introducing component-based software (e.g. [Crnkovic 02, Szyperski 02]).

However, unlike the components from other industries, software is an intangible product. As put by Szyperski, rather than delivering a final product, software developers deliver the blueprints of a product, or, as he calls it, a *"metaproduct"* [Szyper-

ski 02]. The instantiation of software components may include their parametrization, thus leading to different component instances based on the same component abstraction.

Other metaphors, such as comparing Lego-block building with component-based software development are also common, but they break down when closely scrutinized. Shaw argued that building software from components *"is more like having a bathtub full of Tinkertoy, Lego, Erector set, Lincoln logs, Block City, and six other incompatible kits - picking out parts that fit specific functions and expecting them to fit together"* [Shaw 95a]. The problem is components are built using different architectural styles, and, possibly, to meet different interconnection standards. Of course, one can (and often tries to) choose a particular architectural style and select one software component model, in order to reduce this heterogeneity. But it is often the case that no single architectural style is a perfect match for the requirements of a software system, so the software architect ends up choosing a hybrid architectural style that merges the strengths (and, hopefully, mitigates the weaknesses) of several styles.

The technical implications of the extra flexibility described by Szyperski, and heterogeneity commented by Shaw break the over-simplistic analogies to the notion of components in other domains. We need a definition for software component.

The Software Engineering community has struggled to reach a consensus on what a software component is. There is no shortage of definitions for it:

- Bachman *et al.* define software component as *"an opaque implementation of functionality subject to third-party composition and is conformant with a component model"* [Bachman 00].

- Szyperski *et al.* define it as *"a unit of composition with contractually specified interfaces and explicit context dependencies only, which is subject to third party composition"* [Szyperski 02].

- Crnkovic and Larsson define it as a reusable unit of deployment and composition that is accessed through an interface [Crnkovic 02].

- Councill and Heineman define components as software elements that conform to a component model and can be independently deployed and composed without modification according to a composition standard [Heineman 01].

While all these "text-book" definitions overlap on the most fundamental issues, the more intricate technical details may vary. For instance, only two of them make an explicit reference to the component model.

The definitions presented so far for software components were fairly independent of the particular component model. There are also definitions of components proposed in the specification of a particular component technology, or model:

- In Catalysis, D'Souza and Wills define component as a reusable part of software that is independently developed and can be combined with other components to build larger units, and may be adapted, but not modified [D'Souza 98].

- In the UML specification, the Object Management Group (OMG) defines a component as *"a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment"* [OMG 07].

- In the Component Object Model (COM) specification, a component is defined as a piece of compiled software which is offering a service [Microsoft 96].

Finally, it is also common to find object-oriented classes referred to as components (e.g. [Inoue 05] refers to Java classes as components).

A classification of software components proposed by Lau and Wang [Lau 07] helps clarifying how the concept of components is mapped to software. There are three main categories: components may be represented as **classes**, **objects**, or **architectural units**:

- The first category corresponds to components which are represented as special classes, in an object-oriented language. A typical example of this category is that of JavaBeans components [Hamilton 97], which are, essentially, special Java classes.

- The second category corresponds to components which are represented as runtime entities that behave like objects in the component model (e.g. .Net components [2]).

- The third category corresponds to components which are represented as architectural units (e.g. UML 2.0 components [OMG 05b]).

Another possible way of classifying components refers to their granularity, and allows classifying components from fine-grained to coarse-grained.

In summary, the term software component is too overloaded and there is no generally accepted definition for it. Throughout this dissertation, the reader can assume Szyperski's definition as a reference. Szyperski's definition is cited more often in the literature than the alternatives and is neutral with respect to several possible component classification taxonomies, such as the component representation and granularity. The definition by Crnkovic and Larsson is more minimalistic than Szyperski's in its most simple form (the one reproduced here) although the Crnkovic and Larsson also present a set of desirable, but not mandatory, extra requirements on components. Both the definition of Bachman *et al.* and that of Councill and Heineman are stricter than Szyperski's, by requiring a component to be defined according to a component model. While we regard this property as highly desirable in a CBD approach, we do not consider it mandatory.

[2] `http://www.microsoft.com/net`

The remaining definitions presented in this section either have a more specific technology oriented background (e.g. Microsoft's COM components), or refer to an earlier notion of what software components are (e.g. McIlroy's reference to a component industry that would provide libraries of reusable functions).

2.2.1 Software components specification

Components and interfaces

Figure 2.1 depicts the basic concepts concerning components specification using a simplistic UML metamodel, adapted from [Lüders 02].

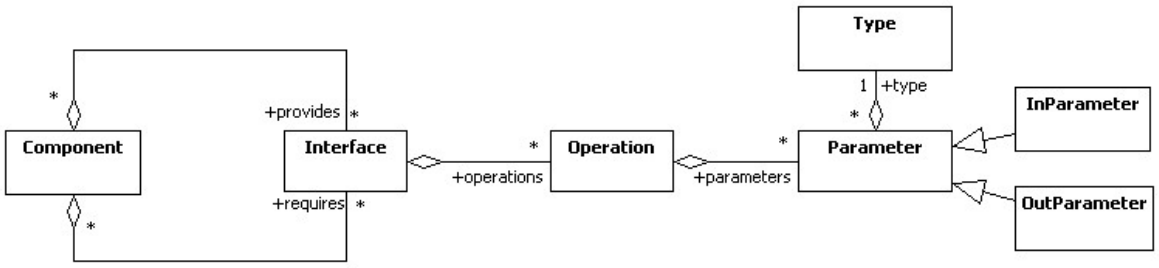


Figure 2.1: Basic component specification concepts

A component exposes its functionalities by providing one or more access points. An access point is specified as an **interface**. A component may provide more than one interface, each interface corresponding to a different access point. An interface is specified as a collection of operations. It does not provide the implementation of any of those operations. Depending on the interface specification technique, the interface may include descriptions of the semantics of the operations it provides with different degrees of formality. The separation between interface and internal implementation allows the implementation to change while maintaining the interface unchanged. It follows that the implementation of components may evolve without breaking the compatibility of software using those components, as long as the interfaces and their behavior, as perceived by the component user, are kept unchanged with respect to an interaction model. A common example is to improve the efficiency of the implementation of the component, without breaking its interfaces. As long as that improvement has no negative effect on the interaction model between the component and the component clients, and the component's functionality remains unchanged, the component can be replaced by the new version.

A component may externalize its set of provided functionalities through its set of **provided interfaces**.

In order to operate correctly, a component may require the existence of a set of interfaces to be available in its surrounding environment. Such interfaces can be provided by other components. These interfaces are often referred to as **required interfaces**.

An operation is specified along with its set of typed parameters. In the metamodel represented in figure 2.1 parameters can be specialized into input or output parameters. In this metamodel, we consider the return value of a function as an output parameter.

Contracts

An interface specification can be viewed as a contract between the provider of the interface and the interface's client [Szyperski 02]. A **contract** establishes the obligations and benefits of each of the contract's partners. An interface defined in a programming language such as Java [Gosling 96] can be regarded as a contract between classes implementing that interface and classes using the interface: a class that implements the interface is obliged to fulfill the interface specification while the interface client provides arguments of appropriate types [Plösh 04].

Depending on the expressiveness of the component's interface description language, the coverage level of the contract may vary significantly. Beugnard *et al.* identify 4 levels of contract support, ranging from non-negotiable contracts to dynamically negotiable contracts [Beugnard 99]:

- **Syntactic level contracts**, such as the one used in typed programming languages as well as on interface description languages.
- **Behavioral level contracts**, where invariants, pre and post conditions can be defined and checked.
- **Synchronization level contracts**, concerning issues related to distribution and concurrency issues.
- **Quality of service level contracts**, address other non-functional properties of the components, such as performance or reliability.

Meyer [Meyer 00] uses a different taxonomy of four levels, maintaining the first two levels, suppressing Beugnard's third level(synchronization) and breaking down Beugnard's fourth level into two: performance contracts and quality of service contracts. Although Meyer's taxonomy also has 4 levels, we consider performance contracts to be part of the quality of service contracts. We will follow Beugnard's taxonomy to guide our discussion on contracts, as it is, in our opinion, more relevant than Meyer's, in the sense that it explicitly considers the synchronization issues that emerge when distribution and concurrency are considered.

Syntactic level contracts

The basic level of contract support is the type checking mechanism of the programming languages used in the development of the components and component-based applications. On a single programming language environment, this is achieved through the

language's type-checking mechanism. With multiple programming languages, a common interface description language has to be used, so that pieces of software developed in different languages can interact.

The coverage of the syntactic aspects of a contract is far from ideal. Developers cannot safely reuse components without understanding their semantics. This problem, common to other software development approaches, is crucial in CBD, as components are frequently developed by a third party and reused as black boxes. Relying on the access to their source code to understand the details of a component is usually not an option, either because the component is reused as a black box, or because the effort required for understanding the component's semantics would be prohibitive, when the source code is available. Last, but not the least, even if the code is available and the effort to understand it in detail can be spent, the principles of encapsulation and information hiding advise practitioners against doing so. The usage of components should be dependent on their semantics, but not on a particular implementation of that semantics.

Behavioral contracts

The syntactic level of contracts does not define precisely the effect of executing operations. Behavioral contracts are aimed at solving this shortcoming of syntactic contracts. A common approach to provide support to semantics in contracts is the usage of **design by contract** (DbC) [Meyer 92a]. DbC relies on three basic mechanisms: **pre-conditions**, **post-conditions**, and **invariants**. Pre and post-conditions are assertions that must hold before and after the execution of an operation, respectively. As such, they are defined at the operation level, in specification languages that support DbC. An **invariant** is an assertion that is kept true throughout the life cycle of the constrained model element. In the context of CBD, we can define invariants to constrain the valid states of components. Figure 2.2, adapted from [Lüders 02], adds the semantics to the component interfaces described in figure 2.1.

While some programming languages have built-in support to DbC (the most noticeable example being Eiffel [Meyer 92b]), others rely on language extensions [Cicalese 99], pre-processing-based approaches [Bartezko 01, Kramer 98], additional class libraries [Guerreiro 01], or on the usage of reflection mechanisms [Duncan 98] for DbC support. At a higher abstraction level, DbC is supported in modeling languages such as the UML 2.* [OMG 07, OMG 06b]. The UML 2.0 standard includes the Object Constraint Language 2.0 (OCL) [OMG 03b], a typed, side-effect free, specification language that allows, among other things, to express invariants on a UML model, as well as describing pre and post-conditions in operations. As such, OCL provides UML with support to DbC. Formal specification languages, such as the Vienna Development Method (VDM) [Jones 90, Fitzgerald 05] also support the specification of invariants, pre and post conditions.

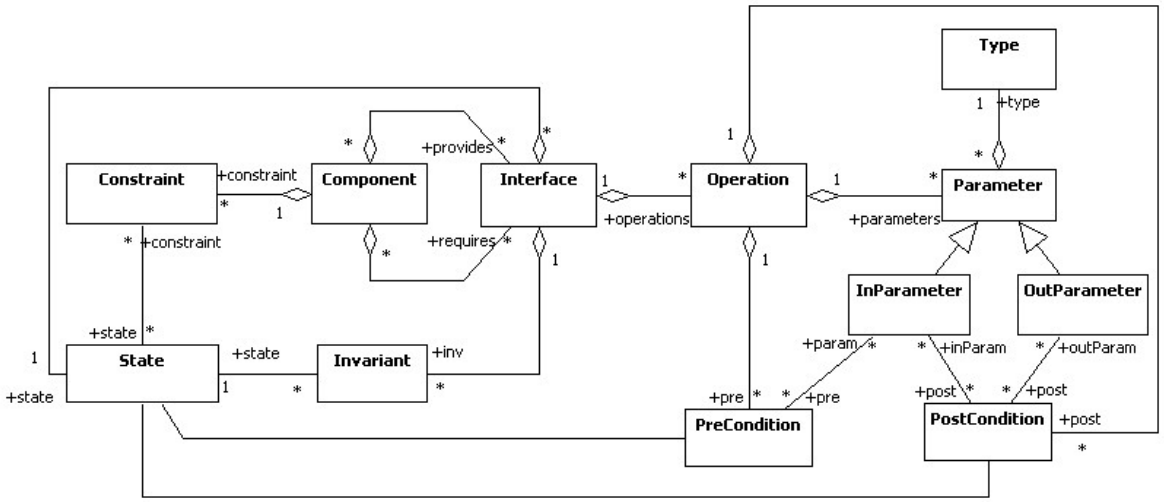


Figure 2.2: Adding semantics to component interfaces

Synchronization level contracts

Behavioral contracts assume that the operations are executed as transactions, which may not necessarily be the case. The third level of contracts concerns the specification of synchronization and concurrency. The aim of these contracts is to describe the dependencies among services provided by a component, such as sequence, parallelism, or shuffle [Beugnard 99]. In other words, they define interaction protocols among cooperating components.

As we increase the sophistication of contracts, we can observe that the direct support for specification of behavior at the interface level, rather than at the implementation level becomes increasingly scarce on mainstream approaches to software development. In modern mainstream programming languages such as Java, or C♯, there is some basic support for synchronization (e.g. through the `synchronized` keyword, used to avoid thread interference). Of course, more sophisticated synchronization policies can also be implemented, but if these concerns are expressed as part of the implementation, rather than directly on interfaces specification, they may become obscure to component users, particularly if the component is being reused as a black box.

There are several proposals for defining synchronization protocols. They are mostly based on formal approaches, such as π-calculus [Milner 92]. By using standard *calculi*, those approaches benefit from the support of the corresponding model checkers to formally derive properties such as liveness, or safety. Some of these approaches use the concept of roles (roles hide the component details which are irrelevant to the particular interaction) to define a modular specification of the observable behavior of components, in the context of their interaction [Canal 03, Li 05]. This corresponds to defining the protocol governing the communication among the components playing those roles, while reducing the complexity that would result from considering the whole components. These protocol specifications are added to the component interfaces definitions.

The protocols are specified with an extension of polyadic π-calculus [Milner 93].

A less formal alternative is to sacrifice the power of process calculus in exchange for notations such as UML's sequence diagrams, which are easier to grasp by common practitioners, but are often used to denote a simplified overview of the behavior, or common use cases, rather than complete interaction protocols between components.

Quality of service (QoS) contracts

The fourth level of contracts concerns non-functional properties (other than distribution and concurrency properties). Again, this is far from being a solved problem by the research community. For instance, consider the specification of the efficiency of a software component, when performing a given task. Such efficiency may depend, to a large extent, on the environment under which the component is expected to operate. Although benchmarking information may be obtained, this may not be sufficient for component clients.

Consider the following example: a client component with strict real-time constraints requires the services of an off-the-shelf server component for a given task to be executed in a time period smaller than a fixed threshold. Now, suppose the producer of the server component deploys an upgrade of its component that adds new services at the expense of a small efficiency loss. The client component may cease to work as expected due to this server component upgrade, if it can no longer use the same service as before within its required time frame. Note that as client and server components may be produced by different organizations, the server component producer may not be aware of that specific client's time constraints, when upgrading its component. On the other hand, the contract information available to the client may not be expressive enough to document this performance loss on the server side in a convenient way. In a distributed environment, where factors such as network overload can interfere with real-time constraints, the problem becomes even more complex.

There are some examples of non-functional properties specification at the component interface level. OMG has two standard profiles to the UML language aimed at addressing real-time constraints [OMG 05a] and Quality of Service properties [OMG 06a]. These profiles extend the basic UML metamodel with meta-classes that allow representing the non-functional properties. The Software Engineering Institute has a research stream called *"Predictable Assembly from Certifiable Components"* that adds an analytic interface to the constructive interface of software components. The term constructive interface refers to our traditional notion of interface (a syntactic interface with an API description). The rationale is that these analytic interfaces provide insight on the inner workings of the components, namely on their non-functional properties. The analytic interface concept is used by Grassi *et al.* to extend an architecture description language (xADL [Dashofy 01]) to support those interfaces [Grassi 05].

More generally, there is a close relationship between the approach of defining con-

tracts aware of non-functional properties and the body of work of compositional reasoning. Compositional reasoning is mostly dominated by work on formal systems, and is based on a divide and conquer approach to system properties prediction: in order to obtain properties about the whole (in the scope of this dissertation, component assemblies), we start by obtaining properties on the parts (software components) and then compose those parts properties to obtain the properties of the whole. A common difficulty with formal approaches to compositional reasoning is their complexity and limited scalability, particularly because the most powerful techniques often require the intervention of an expert user [Berezin 98]. When scale renders current formal approaches unfeasible, an alternative is to use an empirical approach, where component properties are observed, rather than asserted, or proved [Moreno 05].

2.2.2 Component certification

Councill defines third-party certification of components as a method to ensure that software components conform to well-defined standards, in such a way that trusted assemblies of components can be constructed, based on that certification [Councill 01]. One of the interesting points of this definition is the usage of the notion of trust. As put by Shaw, there is a difference between what a component does and what we know it does [Shaw 96]. This gap results from the usually inevitable incompleteness of the specification of a software component.

There are good reasons to accept the fact that most component specifications will be incomplete: it is impossible to anticipate all the properties that will eventually be of interest to any particular user of the component. As new information needs arise, the specification can evolve to meet them. If these evolutions are frequent, it may be unfeasible to have a third party certification up to date with those changes. From a pragmatical point of view, we should also consider the costs involved in the production and consumption of the specification as a constraint. Adding too many details to a specification increases those costs, both from the producer's and the consumer's point of view. The latter will have more difficulty in locating the details of the specification which are relevant to him, as the specification grows.

We can only reasonably expect components to be certified within a certain context, with respect to some of their properties. Even with these constraints, at least for some of the component properties (in particular, extra-functional ones), it is more likely that we can *trust* the components to perform as expected, rather than *knowing* that they will.

In their survey on component certification [Alvaro 05], Álvaro and Lemos note a shift from early works, where most of the research concerning certification was focused on mathematical and test-based models. The limitations of such approaches, particularly for some extra-functional properties, facilitated the creation of a second research vein, that recognizes that testing is not suitable in all situa-

tions (and may even be unfeasible) and transforms the pair <property, value>[3] into <property, value, credibility> [Stafford 01b].

Credibility can be represented through the concept of credentials, originally proposed by Shaw [Shaw 96]. Figure 2.3 illustrates this concept. Credentials may be assigned to components, their interfaces, and even their operations.

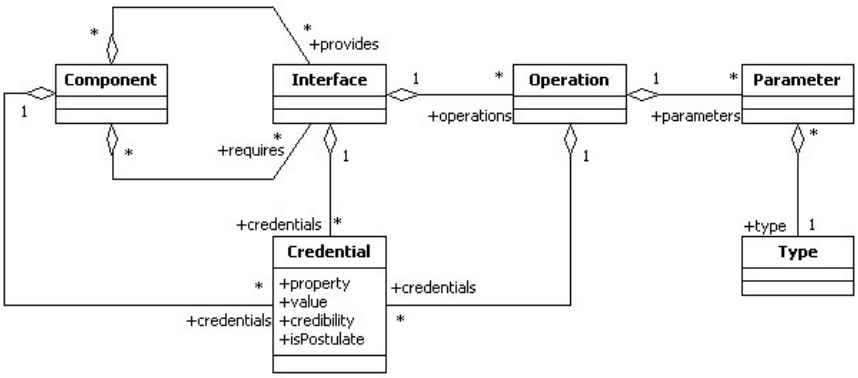


Figure 2.3: Component Credentials

A credential includes an **property**, which is a description of a property of the component, a **value**, which is a measure of that property and a **credibility** indication, which is a description on how the measure was obtained. A partial implementation of Shaw's credentials can be found in UNICon [Shaw 95b]. The boolean attribute **isPostulate** adds to Shaw's notion of credentials, and was introduced in Ensemble [Wallnau 01] to facilitate handling properties for which we have not obtained measures yet, although we need them. If this attribute is true, the credibility is replaced by a plan on how the measure can be obtained.

The diversity of potential extra-functional properties makes the selection of a uniform representation for them a difficult challenge. Although a flexible approach (e.g. using XML) can be used, the added flexibility may, in turn, contribute to the complexity of building analysis support on those properties. Nevertheless, the notion of credentials facilitates building up a repository for storing different tests to the same property, of the same component, each with their own credentials. In the end, the judgment on the merits of the property assessments is still left to the component user, but this approach facilitates the comparison of credentials provided by multiple independent sources, thus contributing to the potential trust users may have on the components they are reusing.

Finally, apart from the technical challenges of certification, it would also be necessary to build up an independent component certification organization that would certify and make those credentials publicly available.

[3]The pair <property, value> stands here for the "traditional" definition of a non-functional property, where a quality attribute, or property, is assigned a value. For instance, one may say that the maximum response time for a given operation is 20 milliseconds, and this information could be conveyed by the pair <maxRespTime, 20>.

2.2.3 Component integration and composition

Integration *vs.* composition

Components are for composition. As such, practitioners have to use a composition language to specify how they want components to be composed. Such a language should provide the syntax and semantics to specify the composition details.

Component integration is the process of "wiring" components so that the needs of a component are satisfied by other components in the integration and vice-versa [Stafford 01a]. The integration often includes not only selecting suitable components, but also defining adapters, so that components which have provided and required interfaces that do not match directly can nevertheless be connected. Although the potential incompatibilities between components are at least partially avoided through the adoption of a component model, which defines standards to support communication between components, mismatches can nevertheless occur. For instance, component producers may have different assumptions concerning how data should be exchanged among components. If the mismatches occur during integration, we call them **integration mismatches**.

Component composition adds to the notion of integration the concern on predicting the behavior of the component assembly, based on the properties of the assembled components. We can think of component integration as the result of using the two inner levels of Beugnard's contracts (syntactic and semantic contracts, including the interfaces, invariants, pre and post conditions), discussed in section 2.2.1. Composition is concerned about the remaining levels of the contracts: synchronization, and the definition of extra-functional properties. For instance, it may be possible to plug (integrate) a set of components in such a way that although there are no integration mismatches, **behavioral mismatches** may still occur. An example of such mismatch is an integration of two components that can provoke a deadlock condition, when interacting together. Another example is an excessive response time of the assembly.

Part of the difficulty with component composition is to select which properties we want to be able to reason about, define techniques for that compositional reasoning, and enacting the specification, measurement and certification of those properties [Stafford 01a]. None of these problems has been thoroughly solved by research in CBSE, yet.

Component integration categories

An orthogonal way of addressing component integration is to discuss the phases of the **component life cycle** during which integration occurs. Lau and Wang proposed an idealized component life cycle with three phases [Lau 05b, Lau 05a, Lau 07]:

- **Design**. In this phase, components are designed, defined and/or implemented, and, if possible, compiled into binaries.

- **Deployment**. In this phase, components are deployed into the target execution environment, where the component-based system under construction will run.
- **Runtime**. In this phase, component binaries are instantiated with data, and these instances are used within the running component-based system.

Figure 2.4 represents the three phases of the idealized component life cycle, adapted from [Lau 07]. Our adaptation considers component integration, rather than composition, as we regard the latter as too restrictive for the state of the art of current component models, as we will discuss in section 2.4.

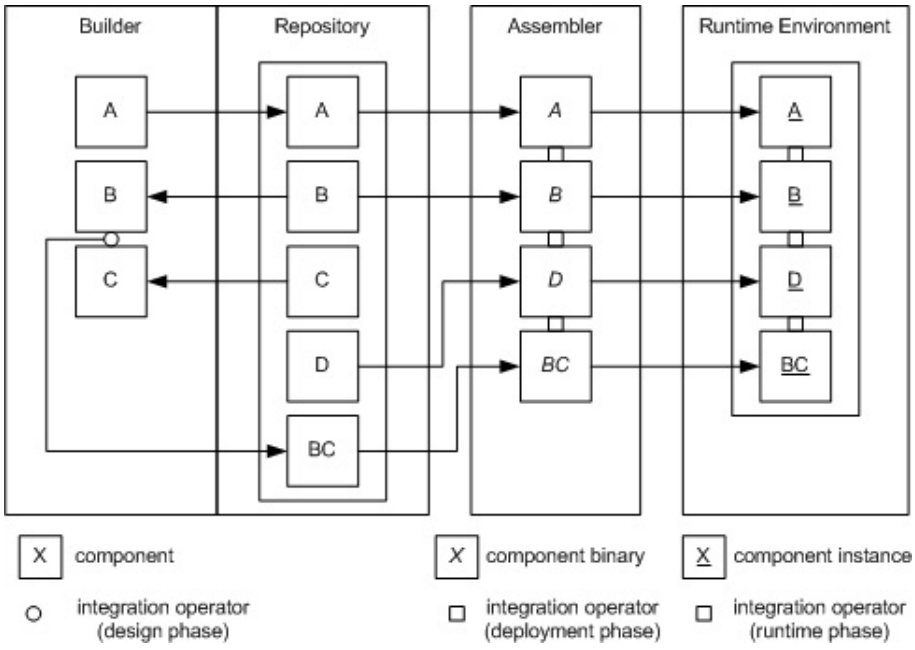


Figure 2.4: Component Life Cycle

The design phase is split into **Builder** and **Repository**, to convey the notion that while some components will be created from scratch and stored in a component repository, other components already exist and can be retrieved from a component repository. The integration of two components results in a new component, in the repository (component BC, in this example). The builder corresponds to the set of tools used in developing a component. The repository is a registry, or a directory, that allows storing, cataloging, searching and retrieving the components.

In the deployment phase, the component assembler tool retrieves the components from the repository, transforms them into binary code, and integrates them into a component assembly.

The component assembly is then instantiated with data in the runtime environment, where the component-based system can be executed.

Depending on the adopted component model, the support to each of these phases may exist, or not. There are four typical combinations of integration mechanisms:

- **design without repository**,

- **design with deposit-only repository**,
- **design with repository**, or
- **deployment with repository**.

The design without repository category, shown in figure 2.5, corresponds to ADLs where it is possible to integrate the components in the design phase, but there is no repository for storing components, and, conversely, for retrieving them (e.g. Acme [Garlan 00b]).

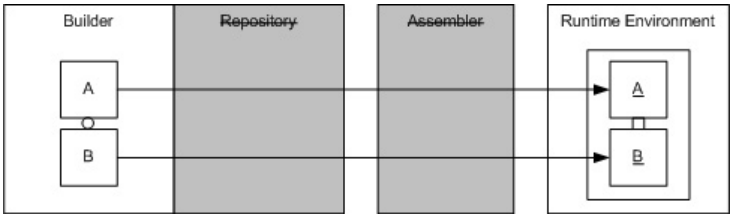


Figure 2.5: Design without repository

The design with deposit-only repository category, shown in figure 2.6, applies to component models where it is possible to design components and store them into a repository, as well as to integrate components at a design stage, although it is impossible to retrieve components from that repository in design time (e.g. CORBA [OMG 02a] components). In this category, it is also impossible to deposit composite components in the repository.

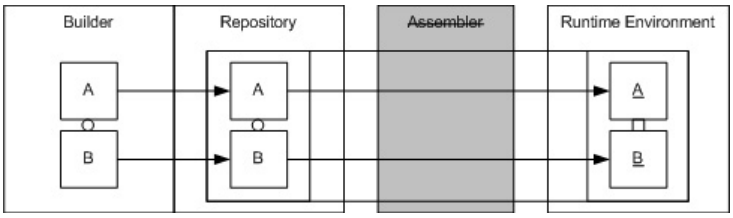


Figure 2.6: Design with deposit-only repository

The design with repository category, shown in figure 2.7 allows for components and composite components to be stored in and retrieved from the component repository (e.g. Koala [Ommering 04] components). As with the other categories, composition cannot be made after deployment and is only available at the design phase.

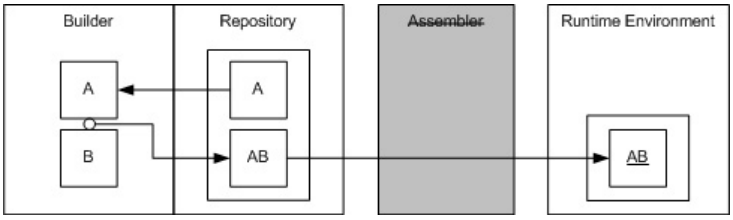


Figure 2.7: Design with repository

The deployment with repository category, shown in figure 2.8 includes components which can be deposited at design time in component repositories, but are only integrated in the deployment phase, when components are retrieved from the component repository and their instances are integrated (e.g. JavaBeans).

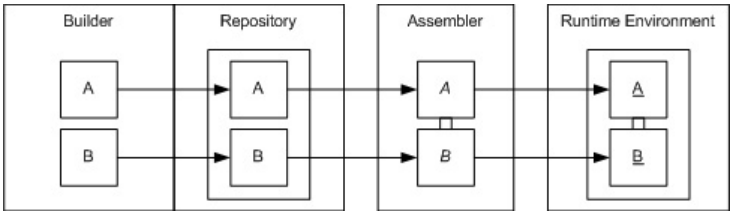


Figure 2.8: Deployment with repository

It should be noted that the component life cycle presented in figure 2.4 would allow other categories here (the ideal one being a component model that allows integrating components in design and deployment, as well as being able to store and retrieve components from the component repository, as discussed in [Lau 07].

2.2.4 Model structure

With respect to the structure of the component model, there are two alternatives:

- **flat**, or
- **hierarchical**.

Flat component models put all components at the same level. In contrast, hierarchical component models allow components to be composed in such a way that the functionality of coarser-grained components can be implemented by finer-grained components, which are encapsulated by the coarser-grained ones. This hierarchical organization of components adds to the flexibility of the component model, with respect to the granularity of components. This enhanced flexibility results from a more powerful mechanism for supporting two important software development principles, **abstraction** and **information hiding**, when compared to the flat model alternative.

2.3 Component-based development process

2.3.1 Fundamental changes from traditional software development

Mature software development follows a well-defined process model. Considering CBD is one of the many possible approaches to software development, it is worth discussing whether or not a generic software development process model is well suited for CBD. Several authors have argued against using a traditional process model in CBD, as described in the next paragraphs.

Ning refers to several aspects of development which are typical in CBD and require special attention, when defining a suitable process model for CBD [Ning 96]. He contrasts CBD with object-oriented development (OOD), where typical development models such as the waterfall model [Royce 70] encourage opportunistic forms of reuse, rather than systematic approaches to it. In such process models, reuse is not regarded as a "first class activity", and it is up to the designers and developers to recognize opportunities for reuse. The lack of a *"de facto"* standard definition for components adds to the lack of systematic reuse by making the identification of potential reuse artifacts harder.

Aoyama identifies several potential approaches to facilitate reuse in OOD, including software architectures, design patterns, and frameworks [Aoyama 98]. All these approaches to reuse are set during development or maintenance. An important contrast from OO reuse to component reuse is that components may have to be composed at run-time, without further compilation, using a plug&play mechanism. This requires components to be viewed as black-boxes, accessible through their interfaces and fosters the definition of architectures for which the components are developed, including the standards for connecting components in those architectures. Component reuse encourages a development process that is concurrent, rather than monolithic, where there are several specialized organizations, as we will discuss in section 2.3.2.

Crnkovic *et al.* add to the discussion the existence of several kinds of CBD, including architecture-driven CBD, product-line CBD, and COTS-based CBD, and argue for the adoption of a process model tailored for each of these varieties of CBD [Crnkovic 06]. The model presented in figure 2.9, illustrates how the CBD process model can be regarded as a combination of several processes that occur in parallel, which may be carried out by independent organizations, to support COTS-based CBD. With some adaptations, Crnkovic *et al.* define variations of this model to support architecture-driven and product-line CBD.

A common point to these three studies ([Ning 96, Aoyama 98, Crnkovic 06]) is the adoption of a modified version of some existing well-known process model (a waterfall model, in the example of figure 2.9), with a shift of focus in some activities and the introduction of parallel process flows for each of the participating organizations. It is also worth noticing the introduction of a third process, component assessment, that can be carried out by an organization independent both from the component developers and component users. Typically, this organization will play the role of a component broker, as discussed in the next section.

2.3.2 Roles in component-based development

With the evolution of the development process to benefit from the reuse of software components, there are a number of novel roles associated to CBD. The roles presented in this section will be used as reference point, when discussing how the quantitative ap-

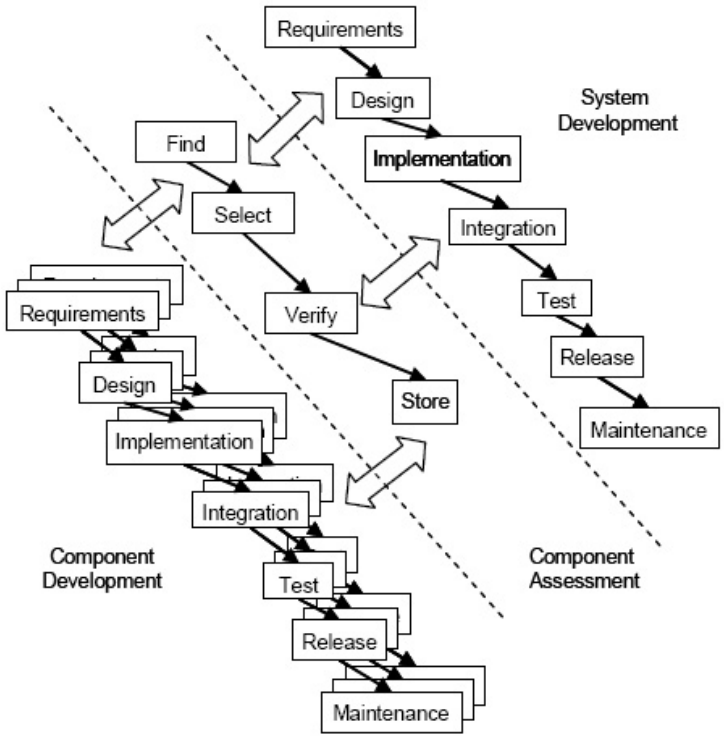


Figure 2.9: CBD process as a combination of several parallel processes [Crnkovic 06].

proaches described throughout the dissertation can aid practitioners performing each of those roles.

Several alternative roles lists have been proposed. Aoyama refers to **component vendors**, **component brokers** and **component integrators** [Aoyama 98]. Aoyama's list is interesting in that it clearly denotes a concern in keeping component producers (vendors) independent from component users (integrators). This separation is further enhanced by the identification of the role of component brokers, who are responsible for selling and distributing software components. Furthermore, in a CBD process such as the one described in figure 2.9, brokers may also play a role in component assessment. For instance, it is common for component brokers to include some form of assessment support for the components they make available (e.g. the Eclipse Plugin Central[4] provides a classification of components, based on user ratings). Different expertise is required for developing and using components, thus leading to organizations specializing in one of the two roles and creating a market for the brokers to aid component users to find and select components produced by the vendors.

Szyperski presents a slightly different list, including **component system architects**, **component framework architects**, **component programmers**, and **component assemblers** [Szyperski 02]. When compared to Aoyama's list, we note the absence of the component brokers and a more specialized view on the roles of component producers (framework architects and programmers) and the component users (system architects and assemblers).

[4] http://www.eclipseplugincentral.com/

The separation between component producers and users creates a challenging technical difficulty: how to test the component, both for functional and non-functional properties.

In practice, these distinctions are not always clear-cut, and practitioners from the same organization can play more than one role, depending on the situation. An example of this can be found in the Eclipse community, where organizations are at the same time developing and providing Eclipse plug-ins to the community, while they are also using plug-ins developed by other organizations (thus accumulating the roles of producers and users). There are a few brokers used by the community to make the components (plug-ins) available to other Eclipse users. Some of the practitioners of the community are also active contributors to the evolution of Eclipse as a plug-in platform and perform, as such, the role of framework architects, as well.

2.4 Component models

A **component model** is *"the set of component types, their interfaces, and, additionally, a specification of the allowable patterns of interaction among component types. A **component framework** provides a set of runtime services to support and enforce the component model"* [Bachman 00].

2.4.1 A taxonomy for component models and technologies

When discussing component models, it is useful to have a common taxonomy, to facilitate their comparison. Taxonomy-based analysis fosters a more systematic approach to the description of the models than the one usually achieved through a more traditional, non-structured, narrative review. Evidence collected in the realm of medical sciences shows that narrative reviews tend to lead to more informal and subjective methods to collect and interpret the studies, including selective citation of literature to reinforce preconceived notions [Pai 04]. In contrast, having a taxonomy for characterizing proposals fosters a more objective analysis, partially mitigating the shortcomings of narrative reviews. The taxonomy also helps readers identifying which models are likely to be applicable to their own context, and which are not. The potential for "selective citation of literature" can also be mitigated by establishing clear inclusion and exclusion criteria for the models and technologies under scrutiny.

The approach of using taxonomies in surveys has been advocated by several authors in the context of the Evidence-Based Software Engineering movement [Kitchenham 04, Dybå 05], with the goal of promoting best practices of other sciences concerning the analysis of accumulated evidence, in the context of Software Engineering. Although this sort of approach to reviews is not novel in Software Engineering (see, for instance, [Laitenberger 02]), it has been adopted in several recent reviews published in major journals devoted to Software Engineering in general (e.g. [Sjøberg 05, Lau 07])

and Experimental Software Engineering, in particular (e.g. [Mohagheghi 07, Kitchenham 08]).

We propose the following taxonomy for describing each entry of our survey:

- **Origin**. A brief note on the origin of the component model. For summary purposes, we will classify each model as:
  - **academic**, if the proposal was originated mainly in the academic community,
  - **industry**, if the proposal was originated mainly in an industrial environment, or
  - **both**, if the proposal is the result of joint work from industry and academia.
- **Component representation**. Here, we will use the representation categories discussed in section 2.2:
  - **classes**,
  - **objects**, or
  - **architectural units**.
- **Component syntax**. One of the distinguishing factors between component models is the language used in component specification. We use three major categories concerning component syntax:
  - **object-like programming languages**, for components which are specified in a given programming language (e.g. JavaBeans are specified as Java classes),
  - **programming languages with IDL (Interface Description Language) mappings**, for components which are specified using an IDL and can then be implemented using a specific programming language which supports that IDL (e.g. .Net components can be implemented in languages such as C♯, or Visual Basic), or
  - **ADL (Architecture Description Languages)**, for components defined using an ADL (e.g. UML 2.0 components).

  Unlike the first two categories, components defined using an ADL have then to be implemented using some suitable programming language.
- **Component integration**. As we discussed in section 2.2.3, different component models support different forms of component integration. We identified four typical combinations of integration mechanisms in that section, and will use them for our taxonomy:
  - **design without repository**,

  - **design with deposit-only repository**,
  - **design with repository**, or
  - **deployment with repository**.

- **Model structure**. Here, we use the categories described in section 2.2.4.
  - **flat**, or
  - **hierarchical**.

- **Contract support**. We use Beugnard's classification of contract support [Beugnard 99] to characterize the surveyed component models. As noted in section 2.2.1, we will consider four levels of increasing contract support:
  - **syntactic contracts**,
  - **behavior contracts**,
  - **synchronization contracts**, and
  - **quality of service contracts**.

- **Support for certification**. The availability of a credentials mechanism, such as the one described in section 2.2.2 will be discriminated in this classification item, with a dichotomic scale:
  - **available**, or
  - **not available**

- **Support for compositional reasoning**. The availability of support for compositional reasoning in the component model as discussed in section 2.2.3, will be discriminated in this section. Again, we use a dichotomic scale:
  - **available**, or
  - **not available**

This taxonomy for component models borrows three of its classification dimensions from [Lau 07]: component representation, component syntax, and component integration. In its original form, Lau and Wang call our "component representation" "component semantics" in the sense of *what is meant by "software component"*. We decided to rename this category as we prefer to use the term "semantics" for expressing the sort of contract that each component model supports.

2.4.2 Models summary

Rather than presenting in this chapter a thorough survey on component models, we opted to present here only a summary of that survey. This summary is intended to give readers a grasp of current component models, their main strengths and shortcomings. Interested readers are invited to find further details on each of the surveyed component models in Appendix A.

Table 2.1 summarizes the component models presented in this review. The balanced distribution of models by origin reflects our concern in discussing component models originated by the software industry, academia, and consortia between both. In the columns' header, we have the surveyed component models. In the rows' header, we present the several categories identified in our taxonomy. The **x** matching a category row with a component model column means that that component model fits into the corresponding categories. For most of the criteria, each component model fits into one of the categories in that criteria. The exception concerns the support for contracts. Here, a model may support more than one contract type, and we opted to mark all the supported types, rather than just the most sophisticated one. As we will see, it is not always the case that a model supporting the most sophisticated of Beugnard's contract types supports all the remaining ones.

With respect to the typical granularity of components, most models support the full range, from fine-grained to coarse-grained components. That said, while some models may be better suited for fine-grained components (e.g. JavaBeans), most models are better suited for medium and coarse-grained components, although they also support fine-grained components. One of the common characteristics for supporting coarse-grained components is the ability to create composite components from finer-grained components (see, for instance, Koala components), and we have classified the models with this ability as hierarchical component models.

Depending on the component model, components may be classes, objects, or architectural units. In our sample of component models, this has a direct mapping on the syntax of the component model. When components are represented through classes, they typically have an object-like specification language syntax. When components are run-time objects, they are usually specified through their interfaces, using an IDL. When they are architectural units, they are specified through an ADL.

When analyzed in the scope of the component life cycle, the form of integration of component models, ranging from design without repository to deployment with repository, varies greatly. In one end of the spectrum, we have ADL-based component models, which usually do not rely on a repository for retrieving their components. The implication that this leads to focusing on designing components and component-based systems from scratch, rather than by reusing existing components, is one of the main shortcomings of these approaches.

On the other end of the spectrum, we have design with repository, where the units

		JavaBeans	Enterprise JavaBeans	COM+	.Net	CCM	Fractal	OSGI	Web Services	Acme	UML	KobrA	Koala	SOFA	PECOS	Frequency
Origin	Industry	x	x	x	x	x		x					x			7
	Academy									x				x		2
	Both						x		x		x	x			x	5
Repres.	Classes	x	x													2
	Objects			x	x	x	x	x	x							6
	Arch. Units									x	x	x	x	x	x	6
Syntax	Object-like	x	x													2
	IDL			x	x	x	x	x	x							6
	ADL									x	x	x	x	x	x	6
Integrat.	DesNRep						x			x	x					3
	DesDORep		x	x	x	x		x	x						x	7
	DesRep											x	x	x		3
	DepRep	x														1
Stru.	Flat	x	x		x	x		x								5
	Hierarchical			x			x		x	x	x	x	x	x	x	9
Contract	Syntactic	x	x	x	x	x	x	x	x	x	x	x	x	x	x	14
	Behavior				x					x	x	x		x		5
	Synch.									x					x	2
	QoS							x	x	x				x		4
Cert.	Supported															0
	Not available	x	x	x	x	x	x	x	x	x	x	x	x	x	x	14
Reas.	Supported									x				x		2
	Not available	x	x	x	x	x	x	x	x		x	x	x		x	12

Table 2.1: Component models.

of design can be stored and retrieved from the component model **as units of design**. This is an approach common to software product line component models (Koala and KobrA), and to SOFA (the exception, when it comes to academic-based models in this review, for this particular categorization).

Between the previous two categories of integration, we have the most popular component models within industry, which, in general, allow designing components and storing them in binary form into a repository. A common shortcoming in these approaches is that they don't allow depositing composite components in the repository, which would make them available for further integration, for instance. Most component models allow component integration during design. The exception is JavaBeans, which only allows it in deployment.

As noted in [Lau 07], there is an opportunity for improvement of current component models, as none of the component models (with the possible exception of SOFA 2.0) supports a "Design and deposit category", which would allow composites created during design to be further composed during deployment.

In [Bures 06], the SOFA proponents dispute this classification. They argue that no composition should happen at deployment time. Their rationale is that rather than a deployment phase, we should consider a runtime phase when systems can be dynamically reconfigured. With this adaptation, they claim to meet the *"ideal"* component life cycle proposed in Lau's taxonomy [Lau 07]. In turn, Lau's taxonomy assumes that runtime reconfiguration is achieved by stopping the system's execution and going back to the deployment phase for reconfiguring the system. It also assumes that the creation of links to services at runtime should not be considered as composition, in terms of semantics (as they are not considered, for example, with respect to web services).

Most component models support, by default, only a basic syntax-type contract. It is no surprise that more sophisticated contract approaches are mostly available for component models provided by academics, or *consortia* between academic and industry partners. It should be noted that this state of practice seems to be shifting, in the sense that, increasingly, industry driven solutions (e.g. UML 2.0) are providing more sophisticated support to design by contract.

The quest for component certification remains, to all practical aspects, a dream. None of the reviewed component models provides direct support for component certification, as described in section 2.2.2. Initiatives such as Predictable Assembly for Certifiable Components (PACC) [Wallnau 03] are trying to mitigate this problem. In [Chaki 07], the PACC team supports certification of software component binaries from UML state chart specifications, combining Proof-Carrying Code with Certifying Model Checking techniques[5].

[5]Proof-carrying code *"constructs a proof that machine code respects a desired policy, packages the proof with the code so that the validity of the proof and its relation to the code can be independently verified before the code is deployed"*, while certifying model checking *"is an extension of model checking for generating proof certificates for finite state models against a rich class of temporal logic policies"* [Chaki 07].

Most component models provide no support for compositional reasoning. The exceptions come from academic backgrounds, suggesting that, as with the certification of components, the techniques for compositional reasoning are still in a maturing process.

The community needs more sophisticated abstractions than those provided by current component models. Components represented by objects and classes are integrated through message passage. Several of the architectural-based approaches discussed in appendix A provide some level of compositional reasoning support, but are less disseminated in the practitioner's working environments, and typically lack professional development tools, comparable to those of modern integrated development environments.

2.5 Metrics for component-based development

With an increasing number of component-based architectures relying on black-box software components [Bass 01], the quality of such architectures depends, to a large extent, on the quality of the integrated components and on the interactions among them [Simão 03]. Therefore, components evaluation should be integrated in CBD [Brownsword 00]. A component assembler takes application requirements, searches component repositories for selecting appropriate components and assembles them, by providing the required glue [Szyperski 02]. His focus of attention is component composition rather than component construction. For the component assembler, it is important to assess the complexity of alternative component assemblies, integrating components that may be acquired from different providers. Deciding whether to reuse components or to develop the corresponding functionality from scratch is also part of his tasks.

In this context, it would be helpful for a component assembler to have an integrated view of existing techniques that may assist him in this task. As pointed out in [Kitchenham 04], Empirical Software Engineering research, in general, tends to be fragmented and not properly integrated. This leads to the absence of a culture of replication of experiments and of systematic reviews of the existing approaches, like, for instance, is common practice for medical research [6]. Therefore, although metrics-based approaches to component reusability assessment have been proposed, to the best of our knowledge, there is a lack of comparative reviews of such proposals. In this section, we provide a comparative study on component reusability evaluation proposals.

The notion of measurement cross-cuts the experimental process in several points. When planning an experiment, we have to define clearly what we need to measure and how we will perform it. The latter implies that our definition of desired measurements is constrained not only by our information needs, but also by the feasibility of performing those measurements, during the execution of the experiment. Furthermore,

[6] For a discussion on systematic reviews in the scope of medical research see, for instance [Pai 04].

our measurement approach should also be designed to facilitate the replication of the experiment by our peers.

The sub-area of Software Engineering concerned with measurement has received considerable attention in the last decades from the Software Engineering community as can be ascertained from its inclusion in the IEEE Software Engineering Body of Knowledge (SWEBOK) [Abran 04] or in the ACM Computing Classification System [7], under the term *metrics*. As stated on the SWEBOK, effective measurement has become one of the cornerstones in organizational maturity. This is further acknowledged by the inclusion of measurement as a key process in maturity models used in the assessment of software development organizations, such as the CMMI [Chrissis 03], SPICE [ISO15504 98], or OOSPICE [Henderson-Sellers 02].

In spite of the wide acknowledgment of the role of measurement in software process and product improvement best practices, practitioners still face difficulties when implementing a measurement program. There are several obstacles delaying a wider adoption of measurement in Software Engineering current practice, including social and technical difficulties [Tichy 98, Abreu 01a]. In this chapter, we are particularly interested in the technical challenges concerning metrics definition. Those challenges span from the specification to the collection and validation of metrics in a production environment. In order to gain a insight on these challenges, we start by observing the main characteristics of current metrics proposals for CBSE.

2.5.1 Metrics and their underlying context

The lack of a widely accepted quality model for CBD is the first challenge for a component assembler in his component selection process. There are some proposals of quality models for CBD, such as [Bertoa 02], where an adaptation of the ISO9126 [ISO9126 01] for component software is proposed, but none of these proposals have achieved an industry-wide acceptance, yet.

Often, metrics definition is not performed to meet the information requirements of a particular quality model, but rather in an ad-hoc fashion. In the absence of such a reference model, interpreting measurements is troublesome.

Consider the example of Lines of Code (LOC) measurement, which could be used in the assessment of the effort required to build white box components. This would be useful when comparing the cost of acquiring a component, *vs.* the cost of building it from scratch. If we simply define how to count the LOC with no reference to how we plan to use them, we are in fact only defining a measurement, but not a metric. Defining the latter implies referring to a framework (the quality model) upon which we plan to interpret the measurements. In other words, even if we are able to estimate that a component will have 20000 LOC, is this good, or bad? How much effort does it take to build such a component?

[7] `http://www.acm.org/class/`

The LOC measurement has been used in several contexts. As a size (or complexity measure), LOC has been used, among other things, to assess the effort required for developing code [Albrecht 83]. One of the problems with this approach is that we can only compute LOC **after** the code is complete. Factors such as the source code reuse level, the particular kind of reuse, or the coding style, may have a significant impact on the value of LOC. The expressiveness of the chosen programming language has also an impact on the value of LOC. While defining **function points** to allow predicting the final code size (in LOC) and effort (in work-hours) for developing such code, Albrecht and Gaffney detected that the chosen programming language has a noticeable effect on the size of the developed software, so they used a language calibration parameter in their effort estimation model [Albrecht 83].

On the other hand, we might be interested in comparing the work of several practitioners, to assess their productivity, and use LOC, or function points, as an indirect measure for productivity, when combined with the corresponding effort in developing code. The (*naïf*) rationale would be that someone who produces more LOC (or function points) than his peers, with a similar effort is more productive. Unless, of course, his code has a significantly lower quality than that of his peers.

The experimental validation approaches for each of the situations described in this section differ. So, a metric may prove to be useful for predicting effort, but useless in what concerns the quality of the outcome of that effort, for instance.

Our points are that:

- it is not possible to define and validate a metric, without clearly stating what is its intended usage;
- as there is no widely accepted quality model for CBD, the validation of current metrics for CBD is harder; we need, at least, to define a context that we can use as reference for metrics validation efforts, so that we can state that metric X is valid in context Y.

2.5.2 Metrics ill-definition

The metrics ill-definition problem occurs due to several reasons. Metrics definitions are often presented without the corresponding context. By context we mean the underlying ontology upon which the concepts of that context, and their interrelationships are defined. Without such clarification, metrics definitions become subjective, as different interpretations on which those concepts are and on how to perform the measurements are possible. Finally, metrics definitions are performed without an underlying formal approach that uses the previously mentioned metamodel as contextual input. The formal specification of metrics should address not only how the underlying concepts are accounted for and their interrelationships are traversed to collect the required metrics, but also the pre-conditions that must be met to allow the collection of such metrics.

Informally defined metrics

Without clear and precise definitions of metrics, it may be impossible to consistently develop tools to collect those metrics, or to discuss their properties in a mathematically sound way. The usage of natural language is a typical metrics definition problem. One of the first books on metrics for object-oriented design contained natural language definitions for all its metrics [Lorenz 94]. While this may be considered helpful as a first glimpse on the metric's objective, the absence of a formal definition may hamper its systematic and repetitive collection and validation by different researchers or practitioners. Consider the following natural language definitions, borrowed from [Gill 03]:

- *"Component Interface Complexity Metric (CICM): Component interface complexity metric should provide an estimate of the complexity of interfaces. Such a metric could be helpful in improving the systems quality because complex interfaces complicate the testing, debugging and maintenance."*
- *"Component Resource Utilization Metric (CRUM): Resource utilization metric should measure the utilization of target computer resources as a percentage of total capacity."*

The first definition is a typical example of a *"wish list"* metric proposal. Although it contains an intuition to the authors' intentions when defining it, the description is too vague with respect to how the interfaces complexity should be measured. The second definition is more objective, in the sense that it implies that the metric is defined as a ratio between used and available resources. It completely omits which resources should be measured and how they could be measured. For the sake of argument, assume that we wish to instantiate the second metric by computing CRUM considering memory as the resource under scrutiny. Which would be the conditions for performing the measurement? Should we consider the average memory used by the component during its lifetime, its highest value during a particular period of usage, or any other option? Should we consider the total physical memory of the target computer as a baseline, or discount the memory used by other applications, namely the operating system being run by that computer? There are too many points of uncertainty in this kind of definition, leading to points of variation in the implementation of tools for collecting them.

Note that even the apparently trivial LOC definition as a count of lines of code is susceptible to different interpretations, in part due to its vulnerability to coding style options. When analyzing a source code file, should we make a simple count of lines, or should we omit, for instance, blank lines? Should comment lines be counted as well, or omitted? How do we deal with text wrapping? Should we pre-process the source code to ensure a uniform formatting style?

In principle, one can always detail all the counting rules down to their most intricate details. Nevertheless, most natural language definitions of metrics are incomplete and

ambiguous. A consequence is that different tools collecting allegedly the same metric may provide different values for that metric, while analyzing the same artifact. This hampers the comparability of metrics collected by independent teams using different tools. Results interpretation may also be flawed, due to these potentially different interpretations of the natural language definitions.

Metrics defined with set theory and simple algebra

A common approach to increase the quality of metrics specifications is to use a combination of set theory and simple algebra to define metrics. Consider the following example, borrowed from Hoek *et al.* [Hoek 03], for the provided (PSU_X) service utilization metric.

$$PSU_X = \frac{P_{Actual}}{P_{Total}} \tag{2.1}$$

Hoek *et al.* define a service as follows: *"Under the term service, we include such things as public methods or functions, directly accessible data structures, and any other kind of publicly accessible resource one may be able to express in an ADL."* Their intention is to define these metrics in a generic way, so that they are not tied to any particular ADL (Architecture Description Language), or service. The price to pay for this option is that different implementations will consider different kinds of services as relevant. Although the metrics formulas are objective, the selection of the elements to be included in such formulas is ambiguous, making these metrics ill-defined.

Formally defined metrics

An alternative is to use a formal approach to define metrics. Algebraic definitions can be made elegant and sound. For instance, consider the definition of the *countServices* metric, which counts the number of services offered by a component. First, we have to define the basic operations so that a component can be created and a service can be added to it. Then, we can define the *countServices* operation. Informally, we can define these operations as:

- *new - creates a new component*
- *add - adds a service to the component*
- *countServices - counts the number of services in the component*

We can then define operation signatures for this simplistic model.

$new : \rightarrow component$
$add : service \times component \rightarrow component$
$countServices : component \rightarrow nat$

Finally we have to define the semantics of these operations. For instance, the semantics of *countServices* could be defined like this:

s : service
c : component
...
countServices (new) = 0
countServices (add(s, c)) = 1 + countServices(c)

The idea of using algebraic approaches to software measurement has been used by authors such as Shepperd [Shepperd 91], not only for defining the metrics, but also to theoretically validate the metrics by defining axioms that characterize desired properties of those metrics, and then demonstrating that the axioms are invariants of the model defined by the algebraic definitions. For instance, one can define an axiom that states that adding a service to a component will never decrease the *countServices* metric, and then demonstrate that the axiom is an invariant of the model.

The main shortcoming of this approach is its *"user-friendliness"*. Although the usage of algebraic definitions is widespread among the scientific community (even if not in the context of software measurement), most practitioners feel uncomfortable with them, as they are not used to dealing directly with algebras frequently. So, algebraic definitions of metrics are likely to be dismissed by most practitioners, with the pretext of being *"too theoretical"*. The ideal situation is to use a language with a sound algebraic background, but similar to the languages used by practitioners in their routine work, so that this knowledge gap can be bridged.

2.5.3 Insufficient validation

The validation of Software Engineering proposals published in the literature is often insufficient. This is true for the general case, as well as in what concerns the validation of software metrics proposals. We can identify two main threads of research concerning the validation of software metrics: **experimental** and **theoretical** validation.

Metrics experimental validation

In what concerns the experimental validation of metrics proposals, the discussion in chapter 3 is fully applicable to the research area commonly referred to as *"software metrics"*. More often than not, metrics proposals are insufficiently validated, from an experimental point of view. One of the main difficulties to experimental validation of software metrics is the need for automatic tool support to metrics collection. This is a barrier not only for the metrics proponents, but also for peers who want to replicate metrics collection. The latter face an extra difficulty, concerned with the informal definitions used in most metrics proposals, as we will discuss later.

Metrics theoretical validation

Metrics theoretical validation is also often missing from proposals, perhaps due to the lack of a generally accepted framework for validation. Weyuker's properties [Weyuker 88] validation is the most widely used.

Weyuker proposed a set of properties for the assessment of software complexity metrics [Weyuker 88]. Her approach is based on the definition of properties that complexity metrics should exhibit. Consider P, Q, and R as programs. Let $|P|$, $|Q|$, and $|R|$ be their complexity, respectively, as measured by the metric under validation. Let $|P;Q|$ be the resulting complexity of P composed with Q. Weyuker's properties, which we now present both in natural language and formally, are as follows [8]:

1. A metric that exhibits the same value for all programs is useless. It provides no information on any of those programs. In other words, it is to be expected that at least some different programs should exhibit a different values for the same complexity metric.
   $\exists P, \exists Q : P \neq Q \wedge |P| \neq |Q|$

2. There is a finite number n of programs for which the complexity is c. To facilitate the formalization of this property (which was not provided in Weyuker's axioms proposal), let c be a non-negative number. Let S be the set of programs with c complexity, and n the cardinal of the set S.
   $\forall c \in \mathbb{R}_0^+ \forall P : |P| = c \Rightarrow P \in S, \exists n \in \mathbb{N}_0 : \sharp S = n$

3. Different programs P and Q may have the same complexity.
   $\exists P, \exists Q : P \neq Q \wedge |P| = |Q|$

4. Different programs which are functionally equivalent (in other words, perform the same task, as perceived from the outside) may have different complexities.
   $\exists P, \exists Q : P \equiv Q \wedge |P| \neq |Q|$

5. Monotonicity is a fundamental property of all complexity measures. It follows that a program in isolation is at most as complex as its composition with another program.
   $\forall P, \forall Q : |P| \leq |P;Q| \wedge |Q| \leq |P;Q|$

6. The resulting complexities of composing the same program (R) with two different programs of the same complexity (P and Q) are not necessarily equal. Conversely, the complexities of composing two different programs (P and Q) of the same complexity with a third program (R) are also not necessarily equal.
   $\exists P, \exists Q, \exists R : P \neq Q \wedge |P| = |Q| \wedge |P;R| \neq |Q;R|$
   $\exists P, \exists Q, \exists R : P \neq Q \wedge |P| = |Q| \wedge |R;P| \neq |R;Q|$

[8] Note that this formalization is an adaptation of the properties definitions presented by Weyuker in [Weyuker 88]. While some of her properties were presented using mathematical expressions, others were defined, either partly or completely, in natural language.

7. Program's complexity should be responsive to the order of its statements, and hence to their potential interaction. Let P be a program and Q another program such that Q is formed by permuting the order of the statements in P. Assume we name this permutation operation $Perm()$.
   $\exists P, \exists Q : Q = Perm(P) \wedge |P| \neq |Q|$

8. If a program is a renaming of another program, then their complexity should be the same. Assume that the operation $Rename()$ transforms program P in its renamed version Q.
   $\forall P, \forall Q : Q = Rename(P) \Rightarrow |P| = |Q|$

9. The complexity of the composition of two programs P and Q may be greater than the sum of the complexities of programs P and Q. The extra complexity may result from the interaction between programs P and Q.
   $\exists P, \exists Q : |P| + |Q| < |P;Q|$

Weyuker illustrated the applicability of her properties with a set of well-known structural complexity metrics (statement count, cyclomatic number, effort measure, and data flow complexity) and observed that none of them exhibited all the properties. Since their publication, Weyuker's properties have been used to support the theoretical validation of several metrics proposals (e.g. [Chidamber 94]).

Before discussing these properties, we have to stress that Weyuker **did not** label her properties has mandatory. She called them *"desirable properties of complexity metrics"*, and even provided an example where it would make sense for a particular metric **not** to adhere to one of the set of usually desirable properties: a metric that uses identifier mnemonics as an input to compute its value does not exhibit property 8, but can nevertheless be acceptable.

However, these properties are frequently referred to as *"Weyuker's axioms"*, and this has been a source for a long controversy, since their publication until today. Cherniavsky and Smith, who referred to the properties as axioms, recognized that the properties were proposed as *"desirable"*, rather than as *"axioms"*, but claimed that satisfying all 9 properties was a necessary, but insufficient, condition for a *"good"* complexity measure [Cherniavsky 91]. Fenton also characterized Weyuker's approach as axiomatic, and identified serious flaws in it, due to the usage of several incompatible views of complexity [Fenton 94]. Kitchenham and Pfleeger joined Fenton in a critical review of Weyuker's properties [Kitchenham 95]. They assume complexity to be related to structural rather than psychological complexity and challenge properties 2, 5, 6, 7, 8, and 9:

- Property 2 concerns the finite number of programs for which a metric has the same value. By analogy to the Euclidean distance between two points, where an infinite number of pairs of points can have the same distance, this property is deemed unnecessary.

- Properties 5, 6, and 9 imply a numeric scale type, so, in practice, they are too restrictive because they exclude other scale types, namely nominal scales.
- Property 7 was criticized for contradicting the standard measurement practice, in the sense that each unit of an attribute contributing to a valid measure is equivalent. Therefore, although a reordering of a program would not necessarily be correct, or of the same psychological complexity as the original one, this should not reflect on the structural complexity of that program.
- Property 8 is considered unnecessary, given the structural complexity assumption. We would add that the psychological complexity assumed by Weyuker in property 7 is dismissed in property 8, because the impact of a renaming in the psychological complexity is difficult to quantify.

Another line of criticism to these properties concerns the applicability of property 9 to object-oriented systems, particularly for metrics that somehow take into account the mechanism of inheritance. Gursaran and Roy noted that none of the inheritance metrics in two of the most influential metrics sets for Object-Oriented design [Chidamber 94, Abreu 94a] exhibited property 9 and argued for the rejection of that property as applicable for this kind of metrics [Gursaran 01]. The formal proof presented by Gursaran and Roy to support their claim was then challenged by Zhang and Xie, who presented a counter-example contradicting the proof, but agreed that this property should be ignored for this kind of metrics [Zhang 02]. The controversy goes on, as Sharma *et al.* presented two new metrics that aim at capturing complexity in the presence of inheritance, and argued that one of Chidamber and Kemerer's metrics (LCOM - Lack of COhesion in Methods) does satisfy property 9, after all [Sharma 06].

The lack of a widely accepted *"theoretical validation"* framework of metrics and the controversy raised by the most well-known set of properties typically used in that validation motivates our choice for considering *"experimental validation"* as our criterion, when analyzing metrics proposals, in the remainder of this dissertation.

2.5.4 A taxonomy for metrics proposals classification

In our survey of metrics for CBD proposals, originally presented in [Goulão 04d], and later refined in [Goulão 07b], we start by proposing a taxonomy for classifying the surveyed works, and then use it to guide our assessment. This taxonomy includes a set of qualitative characteristics plus a quantitative assessment scheme, based on ordinal scales. The quantitative assessment enforces the required comparability of proposals. Together, the qualitative and quantitative parts provide a basis for identifying the strengths and shortcomings of each proposal. The first four items of this taxonomy aim to provide a very brief overview of the proposals, while the last one aims to characterize each proposal according to its maturity level. The taxonomy's characteristics are as follows:

- **Scope**. This refers to the granularity level and type of artifacts that are the target of the metrics-based assessment proposal. A typical contrast is between coarse and fine-grained components. Another one is that while some components are white-box, others are black-box. The scope definition constrains the assessments that can be performed on components.

- **Intent**. A description of the main objectives of the proposal, to help the reader assessing the extent to which each approach may help achieving those objectives.

- **Technique**. This refers to how the metrics were defined and validated. The metrics definition technique may range from a purely informal description to a formal definition. Several forms of validation of the proposals may have been attempted, both by the metrics proponents and other researchers or practitioners. In metrics proposals, validation efforts range from case-studies that use toy examples and aim at illustrating the metrics definition and collection, to series of controlled experiments performed with real-world examples.

- **Critique**. Here, we provide a qualitative assessment of the most noticeable features of the proposal, including its most interesting aspects, as well as its main shortcomings.

- **Maturity**. The maturity level of the proposal provides a comparison framework based on the usage of ordinal scales to characterize the metrics proposals according to four different dimensions: the underlying quality model, the mapping between metrics and the quality model, the formality of the metrics definition, and the extent to which the proposal was validated.

To assess the maturity of the proposals, we start by identifying a set of rating scales concerning different aspects of metrics-based quality evaluation. For each of those rating scales, we then identify several levels of maturity that will aid us in the graphical depiction of proposals maturity. Table 2.2 presents a condensed view of our maturity comparison taxonomy.

Maturity level	Quality Model (**QM**)	Mapping Quality (**MQ**)	Metrics Definition (**MD**)	Metrics Validation (**LV**)
0	N/A	N/A	N/A	N/A
1	Ad-hoc	Ad-hoc	Wish list	Anecdotal
2	Structured	Rationale	Informal	Small experiment
3	Uncorrelated	Goal-driven	Semi-formal	Large experiment
4	Validated	Validated	Formal	Independent

Table 2.2: A metrics proposal comparison taxonomy

The maturity level is of an ordinal nature, ranging from 0, where the dimension is not available in the proposal (N/A in all rating scales), to 4, where the proposal has reached a high maturity level. It should be noted that a proposal's maturity does not necessarily reflect its potential interest. For instance, a radical proposal in an emerging field may be promising, while not yet evidencing high values across all the aspects of our comparison framework. On the other hand, we will expect that within a reasonable period of time, the same proposal will mature. In the next section, we will present several proposals for metrics-based assessment of reusability in CBD. For presentation purposes, we will use the following maturity mask, where **level** is replaced by the appropriate value for each proposal:

QM[level]; MQ[level]; MD[level]; LV[level]

The **Quality Model (QM)** represents the extent to which the metrics proposals fit into a quality model. For the Quality Model, the identified categories, by increasing level of maturity, represent:

1. **N/A**. The proposal is not related to any specific quality model.
2. **Ad-hoc**. A set of quality characteristics are identified.
3. **Structured**. Quality characteristics are organized, typically in a hierarchy.
4. **Uncorrelated**. Quality characteristics are shown to be independent, to avoid assessing the same quality aspect repeatedly.
5. **Validated**. The quality model is conveniently validated through experiments.

The **Mapping Quality (MQ)** represents the level of integration between the model and the metrics which are chosen to assess quality based on that model. The represented categories are:

1. **N/A**. Metrics are not related to a quality model.
2. **Ad-hoc**. Metrics are mapped to quality attributes in an ad-hoc fashion.
3. **Rationale**. A discussion on the rationale of the mapping is provided.
4. **Goal-driven**. Metrics are defined to answer specific evaluation needs, following an approach such as the Goal Question Metric [Basili 94].
5. **Validated**. Building on the previous level, metrics are shown to effectively fulfill the specific evaluation needs raised by the quality model.

Concerning **Metrics Definition (MD)**, we use the following categories:

1. **N/A**. The proposal is only qualitative.

2. **Wish list**. The authors informally identify the need for a certain kind of metrics, without actually proposing any.

3. **Informal**. A natural language description of the metrics is provided by the authors.

4. **Semi-formal**. Some degree of formalism is used in the metrics definitions. Typically, the metrics themselves are defined through mathematical expressions, but the underlying concepts being measured are only informally specified.

5. **Formal**. A formal definition of the metrics based upon the underlying concepts is provided.

Finally, the **Level of Validation (LV)** is classified according to the following categories:

1. **N/A**. The proposal does not include any example of metrics collection.

2. **Anecdotal**. Anecdotal examples are provided to motivate the usefulness of the proposed metric. Sometimes, they are complemented with some descriptive statistics.

3. **Small experiment**. An experiment is carried out to assess the metrics, with some statistical approach to analyze the collected data, but the sample of analyzed artifacts does not allow inference (conclusions generalization beyond the sample used in the experiment).

4. **Large experiment**. An experiment with a significant sample of artifacts is carried out, with real-world artifacts and adequate statistical analysis.

5. **Independently validated**. Experiments conducted by independent research teams confirm the original proponent's claims.

Our overview focuses on metrics-based approaches that aim at helping component assemblers to choose adequate components. The selected proposals share a concern for assessing, somehow, the reusability of components. For easier reference, the proposals will be identified by the name of their first author, both in their textual description and in the chart with the overall comparison, presented in figure 2.10. A reference to the corresponding papers is provided on the top of each of the proposals review.

We have divided these proposals into two groups. The first one considers the components in isolation. The second relates to proposals that attempt to help assessing components in a given context, which is typically either a component assembly or a component library.

2.5.5 Environment-free component metrics

We start by discussing component metrics proposals that assess components in isolation. In other words, the metrics values are intrinsic to the components, rather than dependent on the context in which the components are used.

Bertoa's quality model and metrics [Bertoa 02, Bertoa 04, Bertoa 06]

Scope.
COTS software.

Intent.
To introduce a quality model as an adaptation of the ISO9126 for component-based development [ISO9126 01]. The adaptation of the ISO quality model consists on assuming that the software will include black-box components and change the quality model accordingly, so that any assessment of reused software takes into account this restriction. A set of metrics to assess the attributes of that quality model is also proposed. Its rationale is that the metrics collection has to be defined considering the information made available by component brokers. While the first attempt at metrics definition covers transversally the quality model, more recent work by the same authors focuses on the usability of components, as perceived by component assemblers [Bertoa 04].

Technique.
Although some of the metrics definitions included mathematical formulas, most definitions were informal [Bertoa 02, Bertoa 04]. In [Bertoa 06], where a validation effort for metrics concerning the usability of components is presented, all metrics definitions are presented in natural language. This presentation is complemented by a metamodel describing the information available from COTS vendors that concerns usability. The metrics set includes metrics on the COTS components and their documentation. The metrics collection requires a strong manual intervention, as several of the metrics are collected from the analysis of the available documentation of COTS components. The validation was conducted in a series of 5 experiments (one of them was a replica conducted by peers) with a total of 68 subjects that were asked to evaluate a sample of 12 COTS components. The first three experiments concerned a subjective analysis performed by the participants on each of the components in the sample. The remaining two experiments consisted on an assessment of component reusability through the analysis of the performance of users on answering objective questions concerning the availability of specific tasks and services in the components that made up the sample. Subject's performance was measured as a combination of

correctness of responses and time required for providing such responses, and was used as an indirect measure of component reusability.

Critique.
By using the information made available by vendors, there are limitations concerning the ability to automate metrics collection, due to the noticeable lack of standards in data presentation by COTS producers and brokers. To overcome this problem, a UML model for the classification of COTS usability is proposed, but populating that model in an automated fashion remains an open challenge. From the original set of metrics [Bertoa 02], some were dropped out due to difficulties in their collection. With respect to the validation efforts, the proponents' attempt to build up a set of experiments was successful in what concerns the replication of the experiment by an independent team, but the small component sample is probably the most noticeable threat to validity of the experiment series.

Maturity.
QM [Structured]; MQ[Rationale]; MD[Informal]; LV[Small experiment].

Gill's quality attributes [Gill 03]

Scope.
Black-box components.

Intent.
To propose a set of guidelines on how to select metrics for black-box components.

Technique.
No actual metrics are defined. Instead, the authors informally present a set of quality attributes that should be evaluated through metrics.

Critique.
The proposal includes an interesting discussion on the focus shift caused by the specificity of black-box components evaluation, as opposed to the evaluation of OO design, or software developed with structured programming, and provides an interesting road map for research in metrics-based component evaluation.

Maturity.
QM[Ad-hoc]; MQ[Rationale]; MD[Wish list]; LV[N/A].

Dumke's metrics for reusability of JavaBeans [Dumke 00]

Scope.
White-box Java Beans.

Intent.
To present a metrics set for reusability of JavaBeans.

Technique.
Informal definition of metrics, relying on access to the source code. The metrics in this set are adapted from other contexts, such as OO design (e.g. percentage of public methods) and structured programming (e.g. maximal McCabe complexity number, for a method in the JavaBean class).

Critique.
The white-box view of components renders this approach inadequate for evaluation by independent component assemblers. The internal complexity of a component method should not be relevant for the understandability of its interface and the component's reusability.

Maturity.
QM[N/A]; MQ[Ad-hoc]; MD[Informal]; LV[Anecdotal].

Boxall's interface textual complexity metrics [Boxall 04]

Scope.
Interfaces of components developed with C, C++, Java or Eiffel.

Intent.
To define a set of metrics to assess interface complexity, measuring aspects of components' interfaces, such as the interface size, number of distinct arguments in operations, level of repetition of such arguments, the commonality in identifiers, identifier's length and the density of reference arguments.

Technique.
Metrics are defined through a set of mathematical expressions, but the elements of such expressions are informally described.

Critique.
The level of detail in the analysis of arguments in the interface is richer than in other

approaches, in what concerns the relevance of naming conventions for component interfaces' understandability. However, this approach does not address other potentially interesting aspects in the interface, such as arguments' complexity.

Maturity.
QM[Informal]; MQ[Rationale]; MD[Semi-formal]; LV[Small experiment].

Washizaki's reusability metrics for black-box components [Washizaki 03]

Intent.
To propose a metrics set for assessing the reusability of JavaBeans. The metrics set is defined in the scope of a quality model for black-box component reusability, considering understandability, adaptability and portability as relevant sub-characteristics. More refined criteria are then defined for each of these sub-characteristics, as well as metrics to assess JavaBeans in the light of such criteria.

Technique.
Metrics are defined as ratios of the effective use of a given sort of interface feature (e.g. BeanInfo class, readable properties, writable properties, methods with parameters and methods with no return value) when compared to its potential use.

Critique.
It can be argued that the analysis of the interface complexity is over-simplistic since at least two aspects are not considered: (i) the complexity of arguments, and (ii) the repetition of argument types. In both cases no distinction is made. Intuitively, increased complexity and variety of argument types would decrease the understandability of the component's interface. Washizaki's metrics set was validated with a case study where the reusability of over 120 components was assessed, both with this metrics set and by a panel of experts. Results show a high correlation between both assessments, indicating that the metrics defined in this set can indeed be used to assess component's reusability. However, our independent assessment, presented in section 4.4 indicates that the metrics are unreliable for components with a small number of features on their interface. Further independent analysis is still required.

Maturity.
QM[Structured]; MQ[Rationale]; MD[Semi-formal]; LV[Industrial experiment].

Gill's interface complexity metrics [Gill 04]

Scope.
Black-box components' interface.

Intent.
Besides the complexity aspects of interfaces' signature, this proposal also considers constraints upon those interfaces, as well as their packaging, to account for different configurations that the interface may present, depending on the context of use.

Technique.
The overall complexity is defined as the weighted sum of the complexities related to signature, constraints and packaging of the interfaces. For each of these aspects of interface complexity, a definition is also proposed, again using weighted sums of features (e.g. events and operations count, for signature's complexity).

Critique.
Although Gill's proposal has the merit of including constraints and packaging complexities on the assessment, it still lacks any sort of empirical assessment. This hampers the ability of the authors to assign values to the coefficients on their definitions, and, more significantly, our ability to assess the extent to which this approach helps common practitioners to choose among alternative components.

Maturity.
QM[N/A]; MQ[N/A]; MD[Informal]; LV[N/A].

2.5.6 Environment-dependent component metrics

The approaches described in the previous section are mostly targeted at the assessment of components in isolation. They rely on the assumption that the quality of software components influences in some way the quality of the assembled system. The apparent conclusion of this would be that a component assembler should always try to choose the best components in order to optimize the quality of the assembled system. This may reveal to be *naïf*, since we should also consider the context in which the component will operate. Determining how well a component integrates with other components in an assembly may lead to an evaluation that is more worthy to the component assembler, than the one made in isolation [Wallnau 02]. This change of scope allows the component assembler to focus on the quality for his target product: the component assembly.

Sedigh-Ali's quality characteristics [Sedigh-Ali 01]

Scope.
COTS.

Intent.
To discuss the requirements for metrics for CB-architectures based on relevant quality aspects. The authors also present a taxonomy on the categories of costs related to software quality, with cost drivers such as quality improvement, low quality prevention, software failures and external costs related to those failures.

Technique.
High level discussion, rather than a concrete proposal.

Critique.
The main contribution of this paper is an interesting discussion on requirements for metrics for CB architectures, measured at a system level, including insights on how to choose relevant metrics. However, this is an exploratory work based on expert opinions alone, rather than on some sort of quantitative evidence to back up the presented arguments.

Maturity.
QM[Ad-hoc]; MQ[Ad-hoc]; MD[Wish list]; LV[N/A].

Seker's coupling and cohesion for CBD [Seker 04]

Scope.
Black-box components and component assemblies.

Intent.
To define coupling and cohesion metrics for CBD.

Technique.
The metrics are defined using an information theory based approach where components and component infrastructures are represented as graphs.

Critique.
This approach adapts the well-known concepts of coupling and cohesion to the scope of CBD. Except for the nodes in the graph being black-box components rather than

classes, the proposal is similar to coupling and cohesion for OO design.

Maturity.
QM[N/A]; MQ[N/A]; MD[Semi-formal]; LV[N/A].

Hoek's service utilization metrics [Hoek 03]

Scope.
Software product lines.

Intent.
To propose a metrics set that allows assessing software product lines based on service utilization. The rationale for their need is that service utilization in product lines implies a degree of optionality among the components that get used in a given configuration. While some services and components will be part of all configurations of that product line, others are optional. Structural variability is also an issue, as the component assembler has to choose among a range of alternative configurations. Product lines are also typically hierarchical, composed of a set of components, each of which with its own internal structure. As noted in [Hoek 03], the combination of the above mentioned constraints violates the assumptions of most structural metrics that the system structure under evaluation is: (i) single - no optionality considered, snapshots of the system are usually evaluated); (ii) fixed - no structural variability, the system structure is assumed to be kept constant throughout the evaluation; and (iii) flat - the implications of the hierarchical decomposition of the system are not considered in the metrics definition.

Technique.
The metrics are defined around the concept of service utilization (the rate of usage of provided and required services of a component). For individual components, metrics are simply ratios of used services (both for required and provided ones), whereas for component architectures which are fixed and flat (assemblies) these ratios are obtained using the sum of used services against the total of available services.

Critique.
Of all the proposals presented in this overview, Hoek's is the one that best fits the notions of architectural components and assemblies' evaluation rather than individual components' evaluation.

Maturity.
QM[N/A]; MQ[N/A]; MD[Semi-formal]; LV[Anecdotal].

Inoue's ranking significance [Inoue 05]

Scope.
Software component libraries. Although the proposal is instantiated to Java class libraries, it is generic and could be used with other sorts of components, from fine to coarse-grained, both white and black-box.

Intent.
To enable the implementation of a Java class retrieval system (SPARS-J) that aids developers finding out relevant classes for reuse through natural language queries. As the results of those queries tend to be too broad, a ranking system is required to sort the search results in a convenient fashion. The approach is inspired by the computation of impact factors of scientific publications (research papers, books, etc.) and the ranking mechanisms used by modern web search engines. Components are ranked with respect to their reuse in an existing software baseline. The most reused components have a higher rank and are thus presented at the top of query results, as they are more likely to be of interest for the practitioner performing the query.

Technique.
The component rank model uses a weighted directed graph representation for components, where nodes represent the components and edges represent the use relationships among them. The weight of each node is computed as a function of the weight of its incoming edges. In turn, the weight of each edge is computed as a function of the weight of its origin node and the number of outgoing edges that node contains. The computation of all these weights corresponds to obtaining a stationary distribution of the Markov chain [Stewart 95] that the underlying graph models.

Critique.
One of the most noticeable features of this approach is that reuse is assessed in terms of the effective reuse of software components, rather than in terms of expected reuse (e.g. predicted from the component interface's characteristics). This means that the metrics are useless from the point of view of a component developer. In turn, they may be very useful for component assemblers, as they help locating the most frequently reused components. From all the presented proposals, this was clearly the most thoroughly validated one. The ranking system is in use in two different companies, where a small case study concerning user satisfaction with the ranking system was conducted. The results were very encouraging, although a larger sample of users would be required to confirm them. More important, the ranking system was tested with a set of about 6100 components, from the JDK 1.4.2, on a first observational study,

and 180000 components from publicly available Java component libraries, collected from SourceForge [9], on a second one. In both cases, the ranking system obtained significantly better results than those of non-specialized search engines. The authors do not specifically present the underlying quality model, although the proposal assumes that leveraging software component reusability is a promising approach to the development of high-quality software.

Maturity.
QM[N/A]; MQ[N/A]; MD[Semi-formal]; LV[Industrial experiment].

2.5.7 Discussion on metrics proposals

Overview

The overall assessment of the maturity of the reviewed metrics sets proposals for CBSE is summarized in figure 2.10. In this chart, from left to right, we present each proposal, identified by their first author. From the front to the back we present each of the analyzed rating scales. On the vertical axis we have the maturity level, as defined in table 2.2.

The overall low level of maturity throughout the several rating scales supports the claim that research in the area of software components quantitative evaluation is still on a very early stage. We can revisit now the three aspects highlighted earlier (lack of an underlying quality model, metrics ill-definition, and the insufficient validation of proposals).

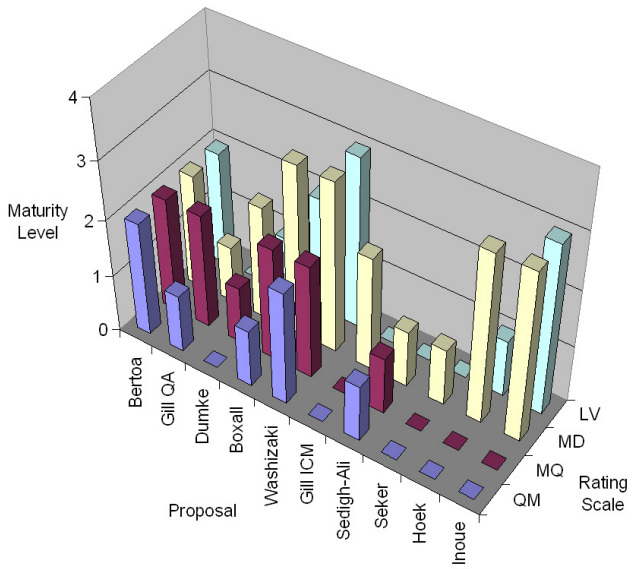


Figure 2.10: Metrics proposals maturity profile

[9] http://sourceforge.net/

Lack of an underlying context

This shortcoming is related to the generally weak relationship among metrics proposals and quality attributes. In the best-case scenario we found proposals where a structured quality model was included, along with a discussion associating the metrics with the quality attributes defined in the model, including the expected effect that variations in those attributes may have on metrics. Washizaki and Bertoa's works were the ones dedicating more attention to this problem, while several other proposals do not explicitly address it in the reviewed publications. This shortcoming of metrics proposals follows a more general tendency observed in other Software Engineering areas, such as that of OO development, where metrics proposals often lack an adequate context [Abreu 01b].

Metrics ill-definition

None of the reviewed proposals includes a formal definition of metrics. In some cases, the author's intentions were clearly to leave the metrics definitions abstract enough for readers to adapt those definitions to their own context (e.g. Hoek's metrics). There is a fairly balanced distribution between wish-lists (3), informal definitions (3) and semi-formal definitions (4) of metrics. Since the majority of definitions are too informal, replicated experiments aimed at validating these proposals are bound to have difficulties related to the tacit knowledge problem: insufficient information provided by the original authors of an experiment causes difficulties in its replication. In this case, the tacit knowledge concerns the definition of the metrics, where non-stated assumptions may lead to different interpretations of the original metrics definitions. While the tacit knowledge problem, as described by Shull *et al.* [Shull 02] is wider, it could be mitigated, in what concerns metrics definition, by providing a formal definition of all the defined metrics. According to Shull *et al.*, the tacit knowledge problem refers to all relevant information for replicating an experiment, from its requirements statement to the results packaging. If this information is not clearly specified in the experiment reporting, those who attempt to replicate the experiment may make wrong assumptions with respect to what was really done in the original experiment.

Insufficient validation

Insufficient validation occurs when independent cross validation is not performed, mainly due to difficulties in experiment replication. Independent metrics validation (not performed by their authors) is fundamental for their proof of usefulness before widespread acceptance is sought.

It is worth noticing that only Washizaki's and Inoue's proposals were validated with industry-level observational studies. Inoue's validation efforts included two case studies carried out in different companies and used significantly larger samples than

any other proposal. It is fair to recognize their validation efforts level are well above the usual state of practice with software metrics. The validation efforts on Bertoa's proposals were also noteworthy for their emphasis on replication, but their main shortcoming seems to be that their metrics collection is partly manual. To the best of our knowledge, the majority of the proposals discussed here were not validated at all.

2.6 Quantitative vs. Qualitative research

Kitchenham classified Software Engineering evaluation types into two main types: evaluations aimed at establishing measurable effects of using a tool, or method, and evaluations aimed at establishing the appropriateness of a tool, or method, to the needs of an organization. [Kitchenham 96a]. The former has a quantitative nature, while the latter is qualitative.

Throughout the experimental work described in this dissertation, there is a deliberate choice to follow a quantitative approach to research. A possible alternative would be to use qualitative research. As noted by Creswell, this would have severe implications in what concerns the whole research design [10].

For instance, in qualitative research, theory and hypotheses are not established *a priori*, in contrast with the practice in quantitative research. So, while qualitative research may help uncovering new research hypotheses, it is not suitable for verifying them. The data emerging from qualitative analysis is mostly descriptive, rather than numerical. While this may be suitable to explain a phenomenon in a given time and location, it is not generalizable. Furthermore, rather than using traditional validation and reliability measures to evaluate qualitative research, it is the believability (based on the coherence, insight, and instrumental utility), allied with a trustworthiness assessment (based on verification) that drive our judgment in qualitative research.

As argued in [Kitchenham 96b], qualitative studies have a generally higher risk of delivering incorrect information, due to their subjective nature. When striving for characteristics such as the generalizability of results and the experimental verification of theoretical claims, a quantitative approach to research is more adequate than a qualitative one.

2.7 Conclusions

This chapter provided the background for the remainder of the dissertation, focusing on two main subjects: the basic notions involved in CBD, and the existing approaches to the quantitative assessment of CBD artifacts (in particular, software components

[10] See [Creswell 03] for an extensive list of key differences between qualitative and quantitative approaches, as well as for an in-depth discussion on qualitative, quantitative, and mixed analysis techniques.

and component assemblies). In other words, we introduced the subject of our research (**software components**), and one of the fundamental tools we will use throughout the dissertation to support CBSE (**software metrics**).

We introduced the basic concepts of CBD and CBSE. We discussed several alternative definitions for the term *"software component"*, as well as their specification, certification, integration, and composition. This initial discussion on terminology showed a wide variety of interpretations for what a software component is. Part of the problem is that the community has not reached a wide consensus with respect to how we should enact CBD. While discussing the software process in CBD, we noted, again, that depending on the particular approach to software reuse, a different variation of the software process would emerge.

All this variability has lead to a plethora of component models and technologies, each favoring its own view of what a component is, and how they should be integrated to build applications. In order to facilitate the comparison between component models, we used a taxonomy to guide our review of such models. The number of existing component models would make it unpractical to analyze all of them in this dissertation, so we chose a balanced sample of the most widely known models, both from industry and academia. Our review of these models concluded the first part of this chapter.

The second part of the chapter is generally devoted to the quantitative assessment of components and component assemblies with the usage of software metrics. We started by introducing the typical problems involved in metrics definitions, and the main shortcomings of current approaches: the lack of an adequate context for metrics definitions, the metrics ill-definition, and the general lack of validation of current metrics proposals. We discussed in some detail the most widely used framework for theoretical metrics validation, and concluded that it has too many limitations to make it suitable for our purposes, thus leading to the alternative of experimental validation of such metrics.

As we had done for component models, we used a taxonomy to guide our review of existing metrics for software components. This review helped us to confirm the previously discussed shortcomings of existing metrics proposals to support CBSE.

Finally, we discussed the main differences between qualitative and quantitative approaches to experimental Software Engineering, in order to motivate our option for following a quantitative approach to support CBSE.

Chapter 3

Experimental Software Engineering

Contents

Background: *Experimental Software Engineering (ESE) is concerned with designing and performing experiments to support the validation of Software Engineering claims. The main challenges of ESE include facilitating the replicability of such experiments and the meta-analysis of the obtained results.*

Objective: *Our goal is to motivate and present a process model for ESE, which we will use throughout the remainder of this dissertation, aimed at solving the above mentioned challenges.*

Method: *We use UML diagrams for describing the process model. The dynamic part of the model is specified through activity diagrams, while a taxonomy of relevant concepts is modeled with class diagrams.*

Results: *The process model conforms to current attempts at experimental reporting guidelines. These results are confirmed not only through our own experience in following the model, but also with a case study conducted with graduate students.*

Limitations: *The case study in this chapter assesses the outcome of the work of graduate students who followed the process. Further validation with seasoned experimenters is desirable, to confirm the merits of this process model.*

Conclusion: *The process model presented in this chapter, and used in the remainder of this dissertation can be successfully used, both by seasoned and novice experimenters.*

3.1 The scientific method

In many research areas, the scientific method is used as a cornerstone set of techniques to collect observable, measurable information that can be used as evidence in the process of understanding a given phenomenon. Based on their perception of the phenomenon under scrutiny, researchers propose hypotheses that attempt to explain the phenomenon and create experiments to test those hypotheses. The results of those experiments are used to test the hypotheses, and, often, to feed back the process, thus leading to more refined hypotheses formulation. A possible description of the method[1], from the formulation of the research question to the publication of research results is outlined in figure 3.1.

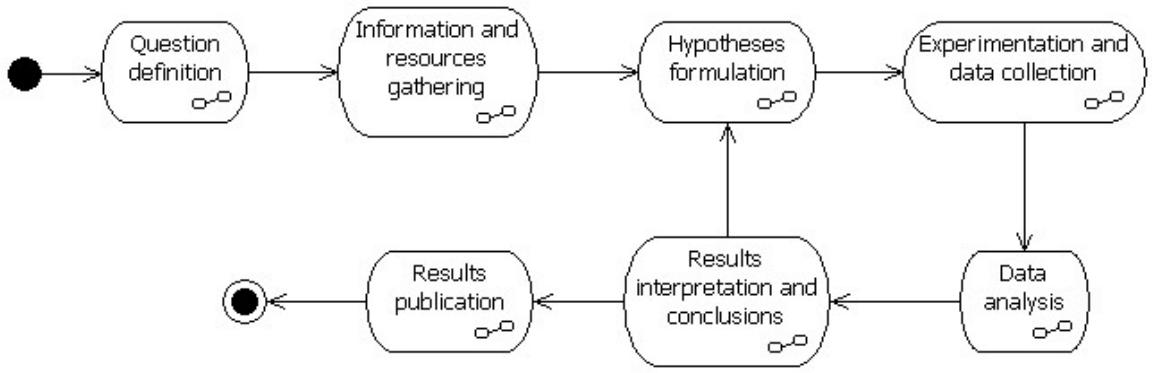


Figure 3.1: The scientific method

The scientific method is designed to reduce as much as possible any potential bias that might be otherwise introduced by the researcher. The whole experimental process is expected to be extensively and unambiguously documented, to facilitate its scrutiny and replication by peers. The level of confidence of the scientific community in the results obtained in this process depends on the level of independent validation the results go through. Knowledge acquired through the scientific method is intrinsically subject to further independent validation, based on experiment replication. Hypotheses and theories are always subject to refinements and even disproof, if new validation efforts point to alternative explanations of the observed phenomena.

This generic description of the scientific method can be easily mapped to the state of practice in most mature sciences, such as physics, biology, or chemistry. Consider the example of clinical trials for the introduction of new drugs in the pharmaceutical market [Vogelson 01]: new drugs have to undergo 4 phases of trials, starting from 2

[1]There are several alternative descriptions of the scientific method available in the literature (e.g. [Koning 94, Wolfs 96, Wudka 98]). They all involve observing a phenomenon, creating hypotheses for explaining it, conducting experiments for assessing those hypothesis, and using the results of those experiments to either support or refute the hypotheses. Experimental results are also fed back to the hypotheses formulation step, so that more refined hypotheses can be formulated.

single-site phases, and ending with 2 phases where between 30 and 40 sites are involved in the trials. The new drugs are often tested against placebos, or other existing drugs, during these trials (particularly on the 3^{rd} phase), and can only be introduced in the market after the first 3 phases. The 4^{th} phase tests the efficacy of the drug for different medical conditions. To remove potential biases, these trials are double- or sometimes triple-blinded. The term **blinding** refers to how much information the patients, researchers, and study monitors have about which particular treatment course a patient is going through. In a **single-blind** trial, the subjects are not aware of information that could bias the results of the trial. In practice, this means subjects do not know whether or not they are part of the control group. In a **double-blind** trial, the experimenters are also unaware of information that could bias the results, so, neither the subjects nor the experimenters know who belongs to the control group and who does not. A **triple-blind** trial is similar to a double-blind trial, but the statistician interpreting the results is also unaware of the treatments administered in the trial. Naturally, the process includes safeguards, so that the blinding can be broken by the researchers in case of an emergency.

We can contrast this state of practice with that of computer science, and Software Engineering. A recent survey by Sjøberg *et al.* on controlled experiments in Software Engineering reported that, out of 5453 scientific articles published in 12 leading Software Engineering journals and conferences from 1993 to 2002, only 103 (1.9%) of them reported controlled experiments where individuals performed Software Engineering tasks [Sjøberg 05]. The authors of the survey define a controlled experiment in software engineering as *"a randomized experiment or quasi-experiment in which individuals or teams (the experimental units) conduct one or more Software Engineering tasks for the sake of comparing different populations, processes, methods, techniques, languages or tools (the treatments)."* [2]

Sjøberg *et al.* counted 14 series of experiment replications. Only 6 of these series included replications performed independently (not by the original authors). 5 out of the 14 series included replications that, at least partially, rejected the findings of the original experiment. Only one of the replications rejecting the findings in the original experiment was conducted by the original team.

The low percentage of experiment-based validation in papers (less than 10 %) [3], when compared to other research methods is noticeable, both in the context of Software Engineering [Glass 02] and Computer Science [Ramesh 04]. These observations are more compelling, when this state of practice is compared with that of other sciences. According to [Tichy 95], the percentage of published papers in computer science that make claims that should require experimental validation support, but pro-

[2] Note that other types of empirical studies, such as studies that are based on observations on existing data, are excluded by this definition. Nevertheless, the insufficient empirical validation of claims is consistently observed in other surveys (e.g. [Zelkowitz 97, Glass 02]).

[3] Unlike in Sjøberg *et al.*'s survey, this 10 % value includes not only controlled experiments, but also other forms of experiment-based validation.

vide none (around 40 % for computer science papers, 50 % for papers on Software Engineering), when compared to papers in other scientific areas, such as Optical Engineering, Physics, Psychology or Anthropology (in these areas, only around 15 % of the papers present no experimental validation), is significantly higher. These results are consistent with the findings of [Zelkowitz 97].

This does not necessarily imply that the Computer Science and Software Engineering communities are producing bad solutions. In some cases, a theoretical validation may be more adequate than an empirical one. But it does limit our ability to assess new solutions, when compared to previous ones. These findings point to an opportunity for significantly improving the state of practice when some sort of empirical evidence is desirable.

Tichy has argued against what he considers the most typical justifications not to perform experimentation, including, among several others, costs, uselessness, and difficulty to conduct experiments [Tichy 98]. In many situations, neglecting experimental evidence on claims leaves other researchers and practitioners with expert's qualitative opinion on those claims. Valuable as such opinions may be, they are based on personal experience and intuition, and thus potentially biased by the expert's background. Either both researchers and practitioners are convinced by the arguments presented by experts, or they are not. But this is a subjective decision to be made, rather than a more rational one, based on verifiable evidence. It is more vulnerable to hype, or fads. When it comes to selecting appropriate tools, languages, processes and techniques, it is desirable to have reliable quantitative facts to support decisions, rather than qualitative opinions alone.

3.2 Evidence-Based Software Engineering

Evidence-Based Software Engineering (EBSE) is a paradigm that supports arguments concerning the suitability, limits, costs, and risks, inherent to Software Engineering tools and techniques, with experimental evidence. The goal of EBSE is *to provide the means by which current best practices from research can be integrated with practical experience and human values in the decision making process regarding the development and maintenance of software* [Kitchenham 04].

3.2.1 The benefits of evidence

Although EBSE is receiving considerable attention from the Experimental Software Engineering (ESE) community, as can be assessed from the calls for contributions of conferences and journals steered by the ESE community, its benefits are yet to be assessed on a wide scale. Nevertheless, the example from other, more mature, sciences, where evidence-based theory validation is a *de facto* standard, can be used as an analogy to the potential benefits of EBSE. The obvious threat to the validity of this argument is

that Software Engineering may be significantly different from other sciences, in this aspect, thus making EBSE inadequate. Kitchenham conducted a detailed comparison between evidence-based medicine and EBSE and concluded that, while the path to EBSE is complex and requires a strong commitment from the research and practitioner communities, it is nevertheless feasible [Kitchenham 04].

There are several successful examples of using evidence in the context of software engineering. Fagan inspections [Fagan 76, Fagan 86] provide some of the most notable examples: for about two decades, Fagan inspections included meetings where potential defects were identified and recorded. However, experimental assessment has shown that there is no significant difference on the number and kind of defects found with meetingless Fagan inspections [Porter 97a]. This lead to the identification of an opportunity to make the code inspections process more effective, considering the cost for detecting a similar number and kind of defects. According to [Tichy 98], Votta reported on significant improvements in the effectiveness of software inspections at a company (Lucent Technologies) as a result of a series of in-house experiments. Other examples include establishing correlations between OO design metrics and software maintainability [Abreu 96] and CASE tools benchmarking [Budgen 03]. Encouraging as these examples may be, they should not be regarded as definitive evidence of the EBSE's benefits. Dybå *et al.* discuss the usability of EBSE for practitioners and report that EBSE is both possible and potentially useful for them, according to early experiences with it [Dybå 05].

If we regard the evolution of the Software Engineering body of knowledge as a quality driven process, the evidence-based approach fits well into it. Both Deming's **plan-do-check-act cycle** [Deming 00] in management, and Basili's **Quality Improvement Paradigm** [Basili 85], for software production, rely on the objective analysis of data collected in projects to detect opportunities for improving the processes under analysis. Following this perspective, Software Engineering proposals should be assessed for soundness. Experimental assessment would be an adequate technique to measure the extent to which new proposals improve existing practices. Note that this is in line with the requirements of the highest levels of maturity in software development, according to maturity models such as CMMI [Chrissis 03] and OOSPICE [Henderson-Sellers 02].

Our point is we have good reason to think that software engineers can achieve benefits by borrowing the know-how accumulated in other, more mature, sciences, that regard experimentation as a fundamental tool for validation of hypotheses and theories. These potential benefits include objective arguments in favor or against current practices, as well as hints that can help driving further research. We have observed such benefits with some Software Engineering practices and, as advocates of experimentation, would like to see this hypothesis further assessed. Note that we are not ruling out the value of expert opinion. We are arguing for the usage of an effective

framework upon which expert's insights can be supported and verified through the collection and analysis of evidence, thus adding to their credibility and contributing to a wider acceptance by practitioners.

3.2.2 The pitfalls of evidence

In general, a single experiment cannot be expected to provide definitive evidence on a phenomenon. Even carefully planned single experiments are subject to threats to validity that may bias their conclusions. Experiments replication can mitigate such threats, particularly by introducing variations to the experimental design. While these replications, also known as differentiated replications, may have weaknesses of their own, the evidence collected in several replications can be used to confirm, or refute, the conclusions of the original experiment. In contrast, close replications try to maintain as much as possible the conditions of the replicated experiment. They are useful both as confirmatory studies and as a means of uncovering previously unnoticed sources of variation in an experiment. In summary, a combination of differentiated and close replications is desirable.

A large enough set of replications can lead to the building of a body of knowledge on a particular subject. However, the aggregation of a body of knowledge from the collection of the lessons learned in disperse independent research efforts remains an unsolved challenge within the Software Engineering community. When confronting the possibly conflicting results of independent studies on the same subject, one needs to be able to weight those results, somehow. A narrative review of studies is helpful to understand those results, but not adequate to draw conclusions on the most controversial aspects of those studies. It is potentially biased by the reviewer's beliefs. Some form of vote counting can be made, but this approach disregards the different strengths and weaknesses of the studies under scrutiny, such as their sample size, or the type of statistical tests applied.

Nearly a decade ago, Brooks advocated that meta-analysis should be used to combine the results of replicated studies, thus solving these problems [Brooks 97]. He also noted the lack of replications to allow for meta-analysis. Miller attempted to perform a meta-analysis on a set of independent defect detection experiments, but found serious difficulties concerning the diversity of the experiments and heterogeneity of their data sets, therefore being unable to derive a consistent view on the overall results [Miller 00]. He presented several suggestions, borrowed from other disciplines, to mitigate the heterogeneity problem. Among those, the development of collaborative efforts for the creation of common repositories for experimental software engineering studies, inspired by examples such as that of the Cochrane Collaboration [4] is a long-term goal with a growing number of supporters within the Software Engineering community.

[4] www.cochrane.org - An international not-for-profit organization which maintains information concerning the effects of health care, based on systematic reviews of existing reports of such effects.

An attempt to bootstrap such a repository was made within the ESERNET [5] European initiative, where several research efforts were coordinated with the intent of gathering a large number of comparable studies The ESERNET repository had, among others, a noticeable feature: experiments could (and should) be registered before their realization, with the introduction of the whole experiment plan, and research hypotheses. The rationale was that this pre-registration would mitigate the **publication bias problem**. In a nutshell, this bias results from the fact that more often than not, only *"successful"* experiments that confirm the hypothesis one is trying to support get to be published. By pre-registering the experiment, it would be possible to assess whether or not the experiment was completed. An experiment is not considered complete until its final report is made available on the repository. Ideally, the results of the experiment should always be made available, even when they did not confirm the expected (*"desired"*?) hypothesis. The rationale is that there is a lot to learn from these *"failures"*, and such information could become very useful for other members of the community. Consider, for example, the cost savings associated with **not** repeating an experiment someone else has shown to provide an outcome different from the *"desired"* one.

Jedlitschka and Ciolkowski presented an overview of the results obtained in the ESERNET initiative, and concluded that, although there was an attempt to coordinate research efforts, thus fostering synergies that would lead to a large number of comparable studies, this effort was insufficient [Jedlitschka 04]. Along with a fairly high mortality of experiments, typically justified by funding problems, the studies were too scattered and too diverse to allow meaningful comparison. Difficulties in aggregating results included:

- the **lack of active coordination** among the independent experiment teams;
- the **non-existence of a roadmap** of required evidence, which, added to the previous point, lead to a high variety of studies;
- the **lack of a suitable method for aggregating the collected evidence**, which made it harder to detect which studies would be required to complement existing knowledge;
- the **lack of guidance for conducting experimentation** lead to a reduced comparability among related studies;
- the **cost of conducting studies** was perceived by the members of ESERNET as an inhibitor for successfully conducting them.

One of the main conclusions of Jedlitschka and Ciolkowski's overview, was that less expensive studies, such as *post-mortems* on existing data, could be useful, even if they have a lower validity than controlled experiments and quasi-experiments [Jedlitschka 04].

[5] www.esernet.org - A network of excellence in the field of Experimental Software Engineering.

3.2.3 Experiment replication and tacit knowledge

If, as stated in the previous section, independent replication of experiments can significantly contribute to the development of a body of knowledge on a particular Software Engineering subject, why are so few independent replications carried out? A (perhaps not that) *naïf* answer would be that researchers are not so keen to perform replications due to their apparent lack of novelty[6]. However, the Experimental Software Engineering community regards replications as essential, and explicitly encourages them in conference series such as *METRICS*, or journals such as *Empirical Software Engineering*. Other, broader-scoped, Software Engineering *fora* such as the *International Conference on Software Engineering* (ICSE) are also supporting this effort. The ICSE'2008 call for papers[7] explicitly requests that *"incremental improvements over previously published work should have been evaluated through systematic empirical or experimental evaluation."* This rules out the possibility of the lack of an audience for such studies.

A deeper obstacle to replication may be its inherent difficulty. A research effort lead by the Software Engineering Lab, in the University of Maryland, provides some insight into this problem. A set of replications of experiments was set up in cooperation with several Brazilian universities. Special laboratory packages aimed at facilitating the replications were built and reused in the different experimental sites, by the different research teams. Shull *et al.* reported that while there was explicit knowledge conveyed by the laboratory packages, there was also significant tacit knowledge that was not captured by those packages and was therefore difficult to transfer from one experimental site and team to the next [Shull 02, Shull 04]. They further argue that this problem is noticeable even with veteran experimenters and well documented experiments.

The obvious approaches to mitigate this problem are to further detail the lab package, as well as to interact with the conductors of the original experiment to fill in the knowledge gaps. Other less obvious improvements may be achieved by including counter-examples, to stress what was **not** done in the experiment. Shull *et al.* argue that a replication process should be added to laboratory packages, and that special care should be devoted to the experiment process conformance.

The existence of a model for experimentation would make the information needs in experiment packages more visible, facilitating replication activities.

3.3 An Experimental Software Engineering process

Solving the problems identified in sections 3.2.2 and 3.2.3 is one of the current priorities within the experimental software engineering community. The publication

[6] In our opinion, this is mostly a problem of perception which results from the lack of a stronger culture with respect to the experimental validation of claims. This *status quo* seems to be changing, as the emphasis on the validation of claims becomes an increasing concern within the Software Engineering community.

[7] `http://icse08.upb.de/calls/research.html`

of guidelines for Empirical Software Engineering research in general [Singer 99, Wohlin 99, Kitchenham 02], systematic reviews [Kitchenham 04], and controlled experiments [Basili 96a, Juristo 01, Jedlitschka 05b] has received some attention over the last few years. While none of these proposals has yet emerged as a community-wide standard, they do convey a tendency toward the comparability among different experiments, through the harmonization of the experimental reports. In this section, we present an extension to these proposals. Our extension adds a process model to the experimental process description. The deliverables of the process model described here are mapped into an underlying logical model of experiment-related concepts that covers the information required by the experiments reporting standard currently being proposed by Jedlitschka and Pfahl [Jedlitschka 05b].

In our model, the deliverables and their relationships are represented with UML 2.0 class diagrams. The activities carried out during the process are described using UML 2.0 activity diagrams. Some of these activities have a direct impact on deliverables of the process, or are fed by deliverables produced earlier in the process. We will use three different stereotypes to decorate the relations between activities and deliverables: `<<read>>` (used when the contents of the deliverable are fed into an activity), `<<write>>` (used when an activity generates a deliverable), and `<<update>>` (used when an activity updates an already existing deliverable, or generates it, if it does not already exist). Note that these stereotypes are not part of the standard UML 2.0 metamodel, but were added using the standard extension mechanisms to increase the expressiveness of our model.

Figure 3.2 presents an overview of a generic experimental process, in the context of software engineering. We will discuss each of the involved activities in the next sub-sections.

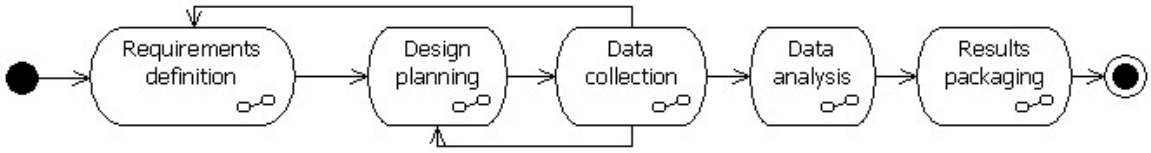


Figure 3.2: Overview of the experimental process

3.3.1 Experiment's requirements definition

From a process point of view, the first step is to clearly state the research problem one is trying to address. The context of the experiment should be defined, as it will constrain, along with the research problem, the definition of the objectives of the experiment. On the other hand, the objectives of the experiment also influence the options made by experimenters, in what concerns the context, hence the feedback loop between these two

activities (figure 3.3). One might expect the experiment's objectives to come first, followed by a context definition to support those objectives. As experiments are typically carried in a resource-constrained environment, more often than not, the objectives of the experiments have to be adjusted to the contexts available to experimenters.

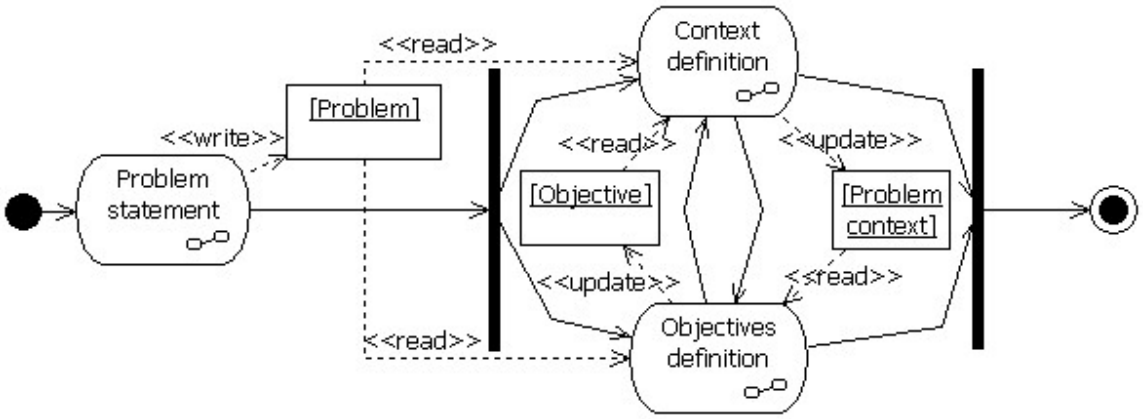


Figure 3.3: Experiment's requirements definition

Problem statement

Software Engineering is a problem-solving discipline. Before conducting experimental work, one should start by clearly defining the problem that the experiment will address, as well as identifying where this problem can be observed (its context), and by whom (the stakeholder who is affected by the problem). Last, but not the least, it is important to state how solving the identified problem is expected to impact on those who observe it (figure 3.4).

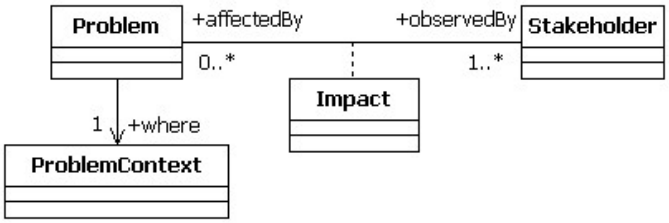


Figure 3.4: Problem statement

Context definition

The context of an experiment determines our ability to generalize from the experimental results to a wider context. Experiments can be conducted in different contexts, each with its own benefits, costs, and risks. These constraints have to be made explicit, in order to ensure the comparability among different studies, and to allow practitioners to

evaluate the extent to which the results obtained in a study, or set of studies, are applicable to his own particular needs. At this phase in the process, an informal assessment of the context is sufficient. This topic is revisited during the experiment design phase, where a more systematic characterization of the context should be performed.

Objectives definition

When conducting experiments, one should clearly define the experiments' goals. Building upon Basili's earlier work [Basili 96a], Wohlin *et al.* proposed a framework to guide the experiment definition [Wohlin 99]. The framework is to be mapped into a template with the following elements: the **object of study** under analysis, the **purpose of the experiment**, its **quality focus**, the **perspective** from which the experiment results are being interpreted, and the **context** under which the experiment is run (figure 3.5).

Wohlin's template for goal description [Wohlin 99] is as follows:

Analyze <Object(s) of study>, **for the purpose of** <Purpose>, **with respect to** <Quality focus>, **from the point of view of** <Perspective>, **in the context of** <Context>.

By providing these informations, the researchers or practitioners conducting the experiment can clearly state why they are conducting the experiment. This is interesting both from the point of view of someone performing the experiment and that of someone using the experiment's results (e.g. in the course of selecting among alternative Software Engineering solutions to a similar problem). The former makes explicit a goal-driven approach to his experimental work. This exercise helps delimiting the experiment's boundaries and focusing on its essential goals. The latter benefits from the availability of a systematic description of experiment goals, which facilitates the assessment of the experiment's relevance to his own context.

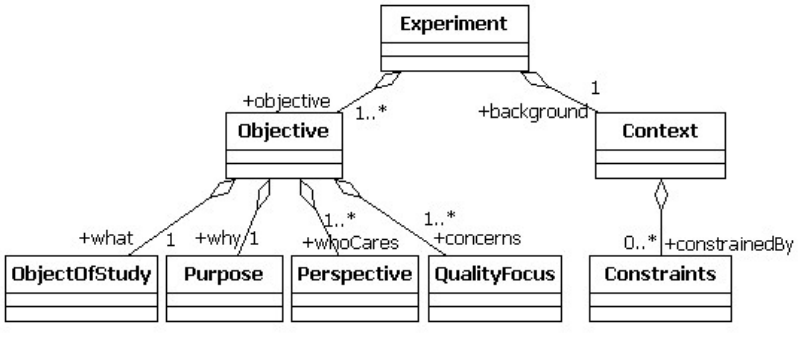


Figure 3.5: Statement of experimental objectives and its context

3.3.2 Experiment planning

While the experiment definition was about **why** a particular experiment is performed, the experiment planning is about **how** it will be performed. Before starting the experiment, decisions have to be made concerning the context of the experiment (revisited here, with more details), the hypotheses under study, the set of independent and dependent variables that will be used to evaluate the hypotheses, the selection of subjects participating in the experiment, the experiment design and instrumentation, and an evaluation of the experiment's validity. Only after all these details are sorted out should the experiment be performed. The outcome of planning is the experimental design, which should encompass enough details in order to be replicable by independent teams. Figure 3.6 describes the activities related to the definition of the experiment design.

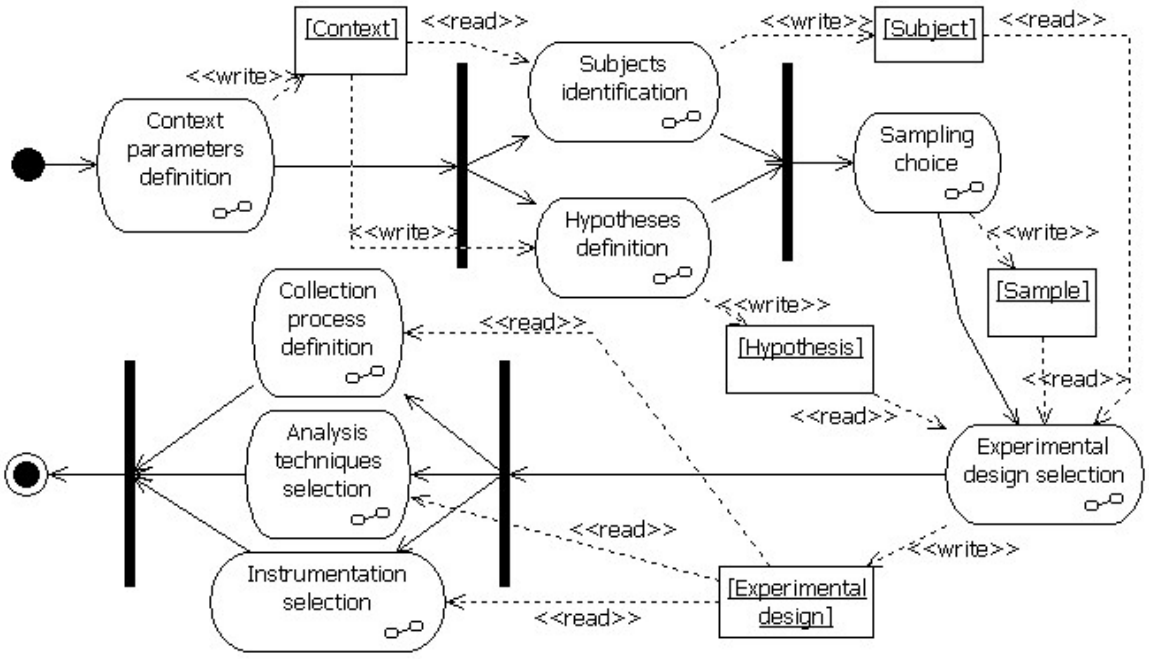


Figure 3.6: Experiment design planning

Context parameters' definition

Throughout the experiment, there are a number of context parameters that remain stable (see figure 3.7). Their value is the same for all the subjects in the experiment during the whole process. Therefore, we can safely assume that differences observed in the results can not be attributed to these parameters. While the actual parameters to be reported may vary, Wohlin *et al.* have identified a core set of context parameters [Wohlin 99].

Concerning their integration within the development process, experiments can be conducted either **on line**, or **off line**. The former, carried as part of the software process

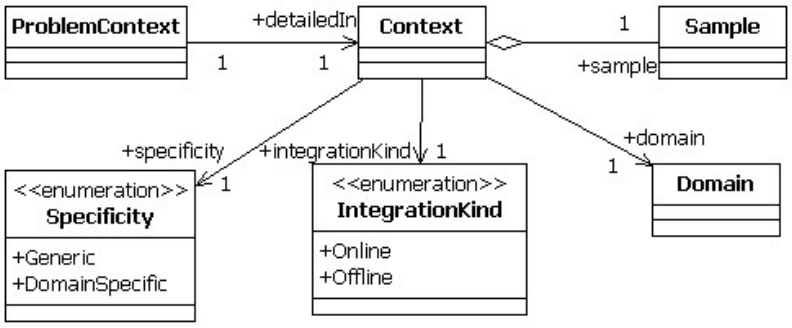


Figure 3.7: Detailed experiment context parameters

in a professional environment, involves an element of risk, since experiments may become intrusive in the underlying development activity. This intrusiveness may even manifest itself through resources and time overheads on a real project. A common alternative is to carry out the experiment off line.

An orthogonal classification of context concerns the people involved in the experiment. One may choose among performing the experiment with professional **practitioners**, or with **surrogates** for those practitioners (typically, students). The first option leads to results that are more easily comparable to others obtained in a professional context, but care must be taken to reduce potential overheads to practitioners' activities (see [Benestad 05] for a detailed discussion on strategies to mitigate some of these risks and thus recruit professional practitioners for participating in experiments).

Using students as surrogates for professional practitioners is less expensive, but makes the experimental results harder to extrapolate for a professional community. To reduce the gap to practitioners the researcher should prefer using graduate students, whose expertise is closer to novice practitioners. A discussion by Höst *et al.* on the using students *vs.* practitioners as subjects and on the circumstances under which students may be used instead of professionals may be found in [Höst 00]. Höst *et al.* carried out an experiment where they assessed the differences between the performance of students and practitioners while performing a non-trivial Software Engineering task. Their overall conclusion was that the differences between students and professionals were only minor and that that students could be used as surrogates for practitioners.

The comparability of results obtained by students and professionals is far from being a thoroughly studied issue. Sjøberg *et al.*'s review [Sjøberg 05] found only 3 papers that compared the performance of students *vs.* that of practitioners. In some of the tasks the results were similar, while on others practitioners did have a better performance. Regardless of the problems that still need to be addressed concerning the comparability between these two groups, performing experiments with students is a valid option for a low cost testing of hypotheses and for educational purposes.

Yet another dimension constraining the experiment is the usage of **toy** *vs.* **real** prob-

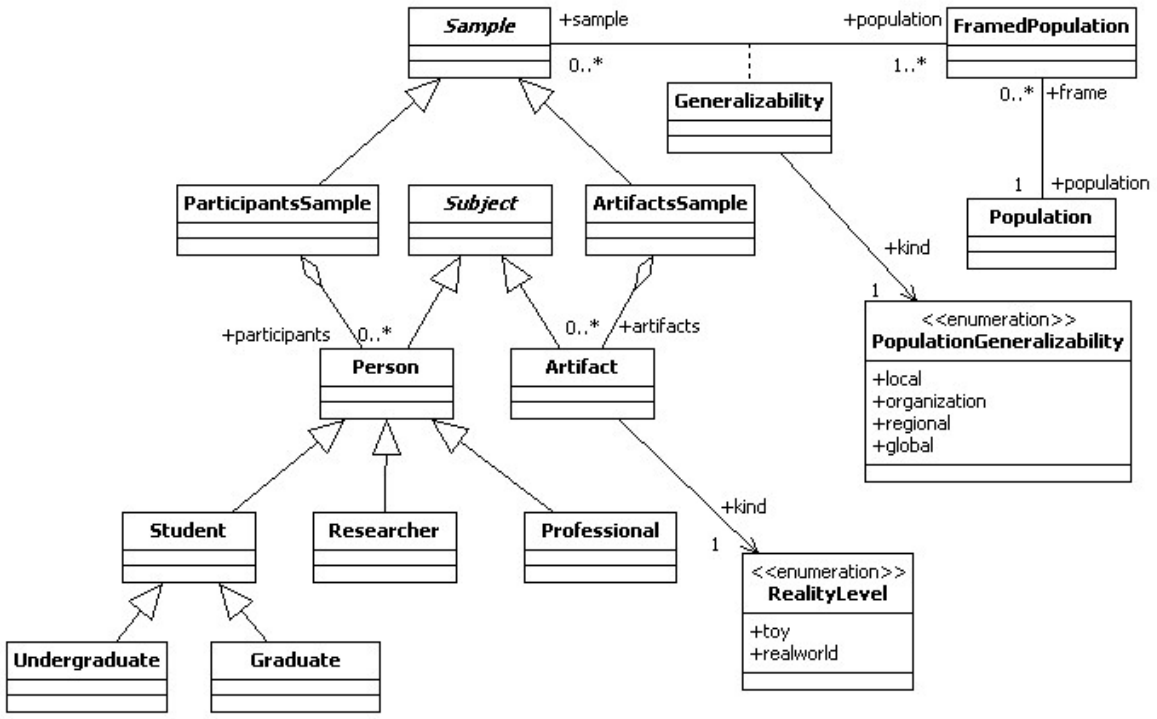


Figure 3.8: Sample characteristics

lems. There are at least two issues that motivate the usage of toy problems: the resources available for the experiment and the risks concerned with the outcome of the experiment. The former results from the, often, very limited time subjects can devote to the experiment. The latter relates to the potential harm caused by the outcome of the experiment (e.g. while experimenting with different testing techniques on a real problem, a less effective technique being tested could lead to a lower final product quality being delivered to a customer). The question, here, is whether the results obtained with a toy problem will scale up to real problems, or not. Toy problems are often used in early experiments, as their usage is less expensive. If the results of experiments conducted with toy examples are satisfactory, the risk of scaling up the problem to a real one may be mitigated to a certain extent, although it will not be completely eradicated.

Experiments can also range from **specific** to **general**, in the sense that their results are applicable to a niche or to a wider population. For instance, when experimenting with the maintainability of object-oriented software, one can design experiments that are language-specific, or experiments that yield results applicable to object-oriented software in general.

Other relevant parameters can be added to this core set. Kitchenham *et al.* argue that context information such as the domain of the software being developed, or organizational constraints such as the development process used by the subjects performing the experiment should also be made clear [Kitchenham 02].

Hypothesis formulation

The hypothesis formulation should be stated as clearly as possible, and presented in the context of the theoretical background it is derived from. This theoretical context makes the hypothesis' implications more apparent, and is important to facilitate the inclusion of the experiment's outcome in the body of knowledge of Software Engineering [Kitchenham 02].

Two hypotheses are formulated: a **null hypothesis**, denoted by H_{0ij}, and its **alternative hypothesis** H_{1ij}. In both cases, i stands for the experiment goal identifier, whereas j corresponds to a hypothesis counter and should be used when more than one hypothesis is being tested for the same goal.

The null hypothesis states that there is no observable pattern in the experiment setting, so any variations found are coincidental. This is the hypothesis the researcher is trying to reject. The alternative is that the variations observed are not coincidental. When the null hypothesis is rejected, we can conclude that the null hypothesis is false. However, if we can not reject the null hypothesis, we can only say that there is no statistical evidence to reject it. Conversely, if we reject the null hypothesis we can accept its alternative. If we can not reject the null hypothesis, we can not accept the alternative.

Hypothesis testing always assumes a given level of significance denoted by α. α represents the a fixed probability of wrongly rejecting the null hypothesis H_{0ij}, if it is in fact true. The probability value (**p-value**) of a statistical hypothesis test is the probability of getting a value of the test statistic as extreme as or more extreme than that observed by chance alone, if the null hypothesis H_{0ij}, is true.

This leads to two types of error that can be made when testing the hypotheses. One can reject the null hypothesis although it was in fact true (**type I error**). The probability for making that error is, as we have seen before, α. One can also fail to reject the alternative hypothesis, although it was in fact false (**type II error**). The probability for making this error, β, is often unknown. Type II errors are frequently associated with samples that are too small.

The power of the test is the probability of not committing a type II error, and should be as close as possible to 1.

Figure 3.9 presents the relationships between the main concepts involved in hypotheses definitions, starting from the overall objectives of the research, through the specific goals of the experiment, and the questions that will allow assessing the achievement of the goals. The hypotheses are then assessed using metrics. The basic concepts concerning variables selection are also included in figure 3.9, and discussed in the next section.

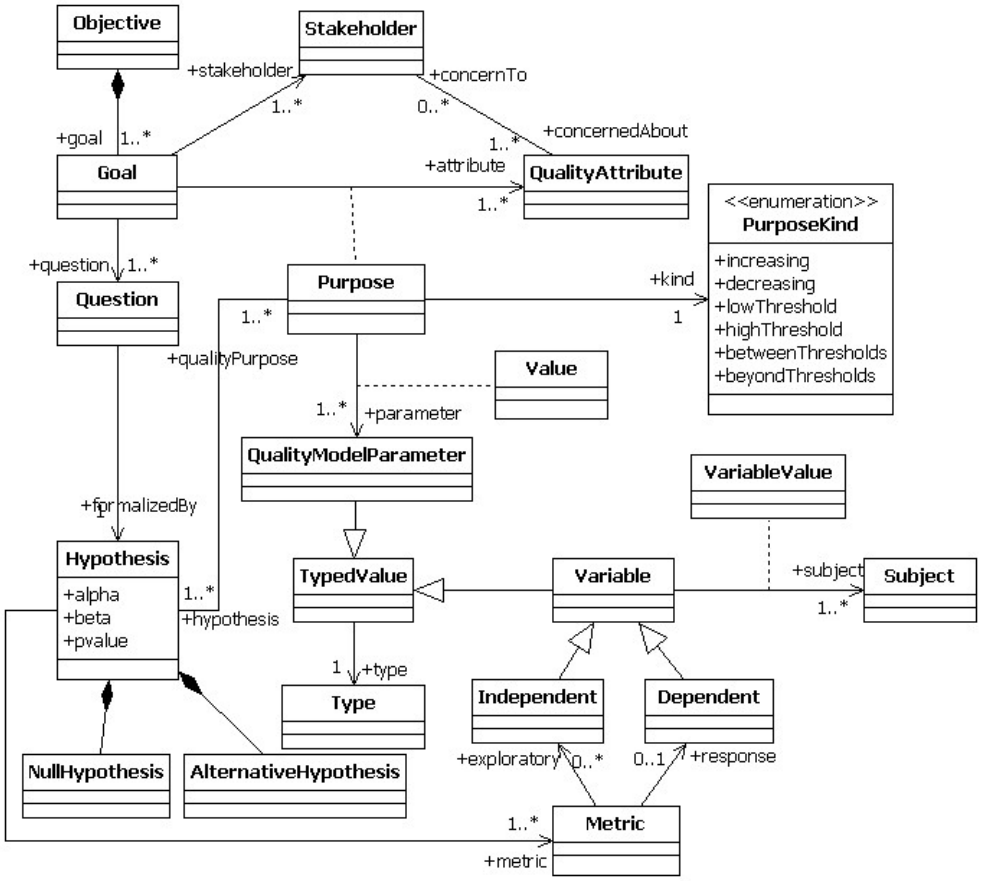


Figure 3.9: Hypothesis specification and variables selection

Variables selection

The process of selecting appropriate variables should be guided by a goal-driven approach, that ties collected information to the research goals that information is intended to help achieving. This way, it is possible to prevent the collection of data that, for the sake of the experiment, is useless, thus saving the resources that would otherwise be employed in such data collection.

In the context of experimental software engineering, the Goal-Question-Metric approach (GQM) [Basili 94] is generally accepted as the standard approach to achieve this objective [8].

The Software Engineering experimenter selects both dependent and independent variables. **Dependent variables** should be explicitly tied to the research goals of the experiment. They should be chosen for their relevance with respect to those goals. When it is not feasible to collect direct measures of the level of achievement of the research goals, surrogates can be used, although such replacement is to be avoided, when possible, and clearly justified, when not. Similarly, **independent variables** are

[8] The GQM approach starts with the definition of a goal, including the purpose of measurement, the object to be measured, the issue to be measured and the point of view from which the measure is taken. The goal is refined into questions which, in turn, are refined into metrics that attempt to help answering them.

chosen for their relevance to the research goals.

Kitchenham *et al.* recommend that, for observational studies and experiments, it may be useful to record additional performance measures that are not directly related with the main research goals of the study, but may nevertheless be affected by the treatments under scrutiny. These extra variables may provide insights concerning possible side effects of the treatments that can be assessed later and motivate further research work [Kitchenham 02].

To facilitate the replicability of experimental and observational work, the variables should be measurable, and, if possible, defined using standard measures. Each measure should be defined as clearly and unambiguously as possible, to prevent different interpretations of its definition. This includes specifying the entity from which the measurement is taken from, the attribute being measured, the counting rule that is applied, and the unit of measurement. We will revisit this subject in detail in chapter 4, when discussing metrics definition techniques.

Subjects selection

The target population has to be defined as clearly as possible. Selecting subjects is not necessarily a trivial task, but it is essential so that:

- the applicability of the results obtained in the experiment is well understood;
- a suitable strategy for selecting subjects can be devised;
- the representativity of the subjects that are selected to represent the population can be assessed (inference ability).

Note that these subjects need not be people. Artifacts, such as software components, can also be used as subjects.

The process of clearly defining the population, in itself, sheds some light with respect to the definition of the applicability boundaries of the knowledge that will be collected with the experiment, with respect to the theoretical framework the experiment is trying to address. Therefore, the population's characteristics, including its invariants, have to be clearly stated.

It is common to use a frame of the population, if it is not feasible to identify all the population's members. In contrast, all members of the chosen population frame are identified. For example, rather than considering all the software components available from any repository for reuse, one can use a frame that considers only the software components available in a known set of components repositories as the population.

Often, it is not possible to perform the experiment using all the relevant framed population as experiment subjects. Instead, a sample of that framed population is chosen, with the objective of being as representative of the framed population as possible,

considering the resources available to the experimenter. So, while planning an experiment, the sampling technique has to be chosen. Figure 3.10 presents a taxonomy of sampling techniques that are applicable to the scope of Experimental Software Engineering.

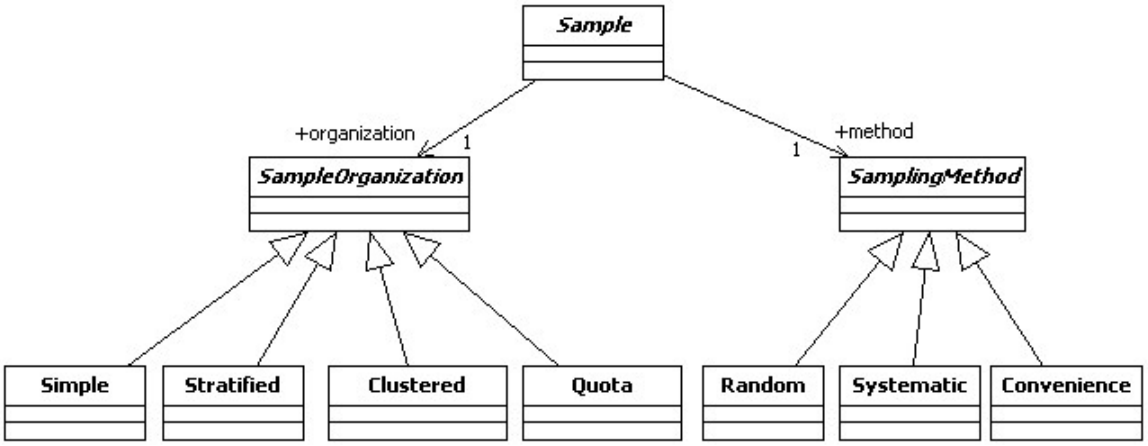


Figure 3.10: Classification of sampling techniques

With respect to the organization, sampling can be:

- **simple** - all elements are treated equally;
- **stratified** - the elements are separated into different categories in such a way that the variations within each categories are minimized, while the variations among different categories are maximized;
- **clustered** - the elements are grouped into clusters;
- **quota** - the elements are grouped into different categories, as in the stratified sample, but then chosen in a non-random way, to ensure a pre-specified proportion among the different quotas.

With respect to the sampling method, it can be:

- **random** - equal probability of choosing any element;
- **systematic** - a rule, such as selecting every i^{th} element in the sample is chosen;
- **convenience** - elements are chosen based on their easier availability.

In the context of Experimental Software Engineering, the most common sampling is a combination of the simple organization with convenience sampling. The experiments reported in this dissertation use this kind of sampling. The implications of this choice will be discussed, in each of the experiment's reports (chapters 6, and 7) included in this dissertation.

Experiment design

The previous choices on hypotheses and variables restrict the available experiment designs. The choice of experiment design is crucial, in that it conditions the valid statistical approaches that can be followed to analyze the data collected in the experiment. Wohlin *et al.* refer 3 general design principles that are used when choosing an experiment design: **randomization**, **blocking**, and **balancing** [Wohlin 99]. Randomization is about averaging out a factor that might otherwise influence the outcome of the test, by ensuring that observations are being made on independent random variables. When the experimenter is aware of a particular factor that may have an influence on the test but is not the factor under test, he may choose to block that effect, by creating several groups within the sample. Within each group, that factor is approximately constant, so that it has no influence on the test being performed. Balancing is about ensuring that each treatment is administered to a similar number of subjects, to ensure a fair test. This is desirable for improving the soundness of the statistical analysis performed during the experiment.

There is no shortage of available experimental design lists, both in the context of Software Engineering (e.g. [Basili 96a, Zelkowitz 96, Juristo 98, Wohlin 99, Juristo 01]), and that of other sciences (e.g. [Cook 76, Creswell 03, Trochim 06]. The recommendations on experimental software practices point to the preferential usage of simple, well-known experiment designs, as they are well documented and can be more easily replicated and understood. In contrast, the usage of custom designs may require the help of a statistician, so that so that the design's implications are well understood [Kitchenham 02].

The criteria for describing taxonomies of designs varies significantly depending on the concerns of the earlier mentioned experimental design lists's proponents. Furthermore, the plethora of available experimental designs is too vast for its inclusion in this dissertation. Rather than providing yet another list of experimental designs, we focus on the basic experimental design building blocks. An experiment design prescribes the division of our sample into a set of groups, according to some strategy. Each of those groups receives a set of interventions, that may be either observations, or treatments. The sequencing and synchronization of such interventions, their nature, and the group definition policy, define the experimental design (figure 3.11).

In experimentation references such as [Cook 76, Creswell 03, Trochim 06], each experiment design is presented as a sequence, or set of parallel sequences of symbols that represent the main constructs of the design: observations (**O**) and treatments (**X**), following a notation proposed by Campbell and Stanley [Campbell 05]. These symbols are decorated with indexes, when different variables, or different treatments are used, respectively. Random assignment of subjects to groups is represented by the symbol **R**. Trochim [Trochim 06] uses two extra symbols for representing non-equivalent groups (**N**) and cut-off groups (**C**), while the original notation used a dashed line to represent

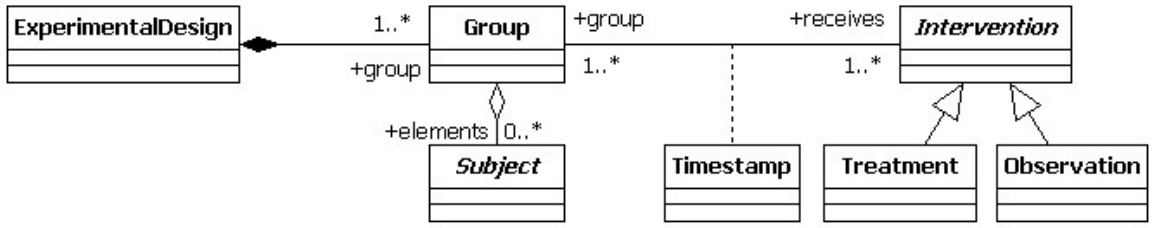


Figure 3.11: Experimental design concepts overview

non-equivalent groups (including cut-offs).

For example, consider an experimental design with two groups, where one group will receive a treatment and the other a placebo (no treatment). Suppose that subjects are randomly assigned to groups. Group A is observed before and after receiving the treatment (these observations are often referred to as pre and post-tests. Group B is observed in the same moments as group A, but does not receive the treatment. This design can be described as:

```
Group A     R   O X O
Group B     R   O   O
```

Note that timing and synchronization issues are represented in this notation by the vertical alignment of the symbols. We can integrate these notions in our process model, by refining the action Experimental Design Selection, referred in figure 3.6. Figure 3.12 presents a first overview of the experimental design selection, where a decision is made concerning the number of groups of subjects participating in the design. With single-group designs, which would correspond to designs with a single line in Campbell and Stanley's notation, the group assignment activity can be skipped as subjects are all assigned to the same group. There are a number of threats to internal validity associated to single-group designs, as we shall discuss in section 3.3.4.

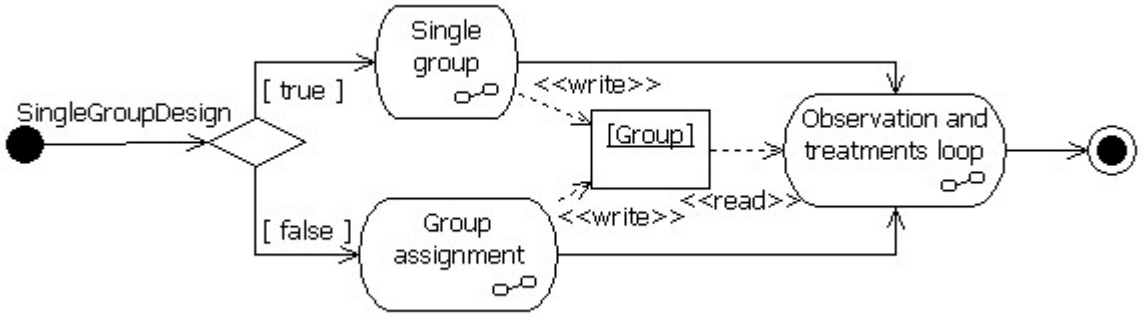


Figure 3.12: Experiment design selection overview

The group assignment is detailed in figure 3.13, where the researcher can decide which of the group division strategies best fits the goals of the experiment.

Random assignment implies that every subject has an equal probability of being assigned to each of the experimental groups. Random assignment is strong against single group internal validity threats, as well as to most multiple group internal validity threats. The latter characteristic stems from the probabilistic equivalence of the groups.

Non-equivalent groups are used very often, in quasi-experiments. A common example is when an experiment is carried out in an academic context and the groups correspond to different classrooms. This implies that the probabilistic equivalence of the groups is lost. It is still often possible to consider the groups to be comparable, and desirable to form groups as similar as possible. Because the groups are non-equivalent, researchers must consider the additional internal validity threat of selection. If the groups are different in a way that affects the outcome of the experiment, this may become a confounding effect to the analysis of the results.

Cut-off groups are used in situations where the experimenter wishes to use a quantifiable property of the subjects as a discriminator of those subjects. The cut-off point between two groups is used as a limit between those groups, so that subjects with a property value below the cut-off are assigned to one group, while subjects with a property value above the cut-off are assigned to another group. This approach is particularly useful if discontinuities are expected between the different groups.

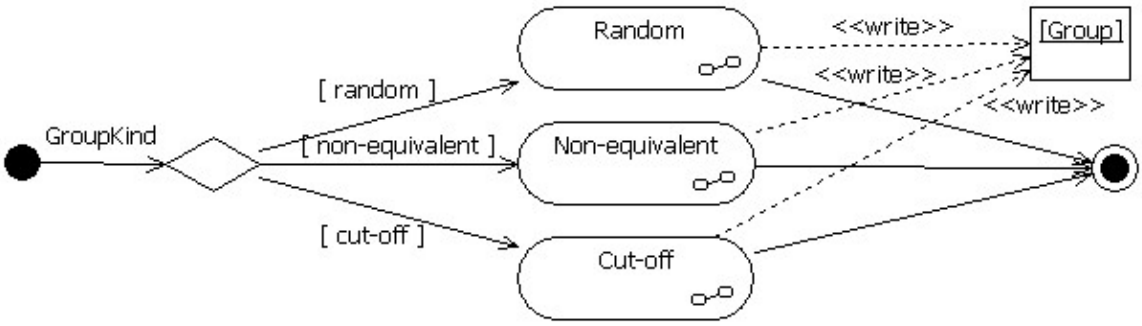


Figure 3.13: Group assignment

Each group is then subject to a sequence of observations and, possibly, treatments, as described in figure 3.14. Note that the sequences always end with an observation, as we assume there would be no point in assigning a treatment to a group and then failing to observe how the group reacts to the treatment, in a well formed experimental design.

A thorough discussion on well-known experimental designs, their strengths and weaknesses (including the inherent threats to their validity) is beyond the scope of this dissertation. Such a discussion can be found in [Cook 76], where Cook and Campbell systematically discuss, providing examples, several designs. Other useful designs can also be found in [Creswell 03]

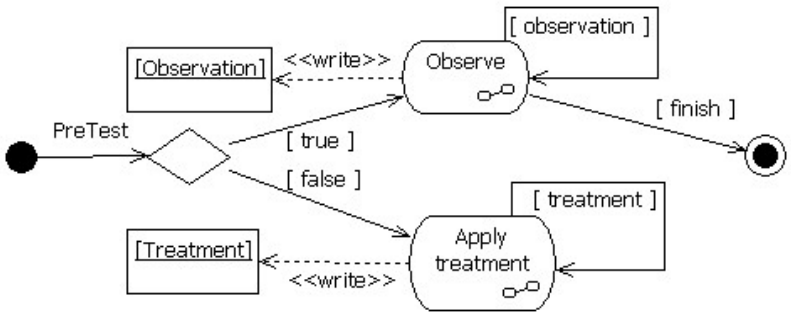


Figure 3.14: The sequence of observations and treatments

Collection process definition

The collection process definition involves planning who will collect the experimental data, as well as when the data collection will take place, and how. In short, a data collection protocol has to be defined. While in some situations the team conducting the experiment will be able to collect the data, in others that process has to be dealt with by the experiment's participants. Data collection activities have to be scheduled, so that potential time constraints can be considered, including the availability of participants. In order to foster the collected data quality, the instrumentation of the experiment will also have to be planned, as we will see in the discussion on instrumentation, later in this section. The driving force should be to minimize data collection effort while ensuring data is collected in a consistent way throughout the process.

Analysis techniques

The analysis techniques chosen for the experiment depend on the adopted experiment design, the variables defined earlier, and the research hypotheses being tested. More than one technique may be assigned to each of the research hypotheses, if necessary, so that the analysis results can be cross-checked later. Furthermore, each of the hypotheses may be analyzed with a different technique. This may be required if the set of variables involved in that hypothesis differs from the set being used in other hypotheses being tested. Figure 3.15 presents a basic categorization of (i) data types, with respect to their **scale** and **level of measurement**, and (ii) of statistical tests techniques. While it is beyond the scope of this dissertation to discuss existing tests in detail, it should be noted that the scale and level of measurement of the variables involved in the statistics tests condition the suitable tests. Discussions relating statistical tests (in particular, parametric *vs.* non-parametric ones) with variable types can be found in statistics text books, such as [Maroco 03].

For example, consider an experiment involving two hypotheses. One of the hypothesis only uses continuous variables, while the other uses categorical ones. Due to the different nature of the used variables, the first hypothesis might use paramet-

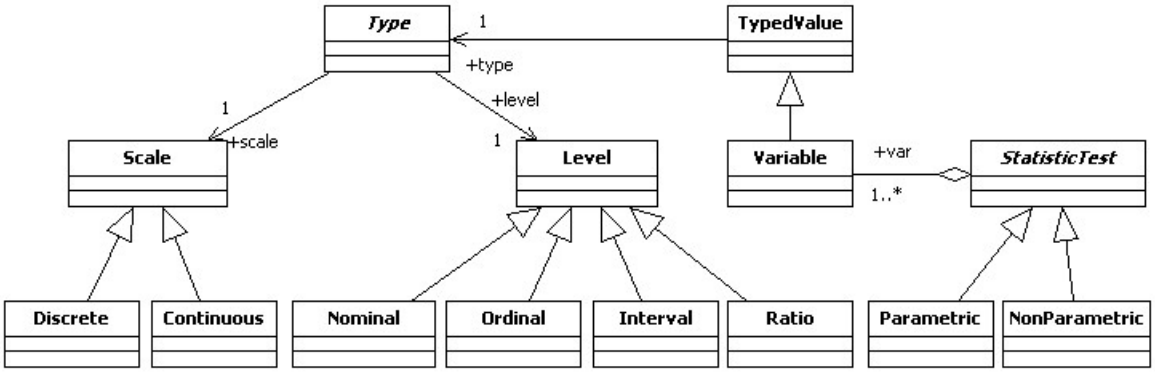


Figure 3.15: Data types taxonomy and statistical tests categories

ric tests (as long as the variables fit a parameterized distribution, such as the normal distribution), while the second one is constrained to using non-parametric tests.

Instrumentation

The instrumentation process involves defining the artifacts that will be used in the experiment. For instance, in a code review techniques experiment, the source code that will be reviewed is an example of an artifact that should be defined in the instrumentation phase of the project. This would probably include not only the definition of the source code artifact, but also that of a checklist of problems seeded in the source code to be found by participants.

The instrumentation also concerns the production of guidelines, and tools (not necessarily computer-based ones) that will support the measurements performed in the experiment. The rationale is to foster the comparability of the collected data by streamlining data collection in a consistent way. Note that instrumentation may also include any training material distributed to participants, before their participation in the experiment.

3.3.3 Experiment execution

The experiment execution is the process where the plan discussed in the previous section is instantiated. The same experiment plan can be instantiated in several different ways, due to local constraints. For instance, both the number of subjects and their exact background may differ, from experiment to experiment, despite sharing a common plan. It is important to document these peculiarities, as they help grasping the "field" constraints of the experiment, in complement to those previously defined in the experiment plan.

Figure 3.16: Experiment data collection

Collection clearance

When the data to be used in the experimental work belongs to individuals, or organizations, collection clearance must be obtained before starting the data collection process. This is particularly important if the data being collected is considered sensitive by its owners. Granting anonymity of participants and organizations is an alternative commonly used by researchers, with the agreement of data owners.

Motivation of participants

The difficulties of recruiting professional practitioners to participate in experiments often lead to the usage of students as surrogates for those practitioners. Sjøberg *et al.*'s systematic review of controlled experiments in Software Engineering [Sjøberg 05] shows that out of 5488 participants in 113 experiments reported on the main software engineering journals and conferences from 1993 to 2002, only 9.4% were professional practitioners, while 86.8% of the participants were students. The remaining participants were either faculty members and post-docs, or of a background not disclosed in the papers reporting the experiment. Although the about 1/3 of the students has an unknown background, one can estimate from the remaining subjects that over 80% of the students were undergraduates.

In general, experiments are framed within a wider context. With professionals, experiments can be performed in the context of a real development project (e.g. a project being used as a pilot for the introduction of a new development tool) or as part of a training course. In most situations, professionals participate in the experiments as part of their job.

In [Benestad 05] Benestad *et al.* discuss the problems concerning the recruitment of professional participants for experiments, and conclude that:

- **Practical constraints** have to be taken into consideration when defining the target populations of experiments. These include geographic constraints, the organizational profile, and the individual profile of participants.

- The participant organizations and individuals have to be offered **flexibility** and **added value**, so that adequate samples of organizations and individuals can be recruited. The flexibility is a facilitator characteristic to ensure that the experiment is as non-intrusive as possible, so that it is not viewed as a burden by the participants. The experiment should also guarantee some form of added value

for the participants. This can range from direct payment (unfeasible, in most situations), to knowledge transfer from researchers to the practitioners, through training sessions, and seminars to share the results of the experiment internally, before making them publicly available. If the experiment shows an opportunity for improving some part of the software process within the organization, this can also be perceived as an added value.

- **High professional and ethical standards** must be achieved, if a continuing cooperation is sought. In the long run, successful experimental work that is found useful both for the researchers and the practitioners involved in it creates opportunities for a continuing collaboration. A typical ethical concern is to ensure the anonymity of participants when some discrimination is to be made with respect to their qualitative assessment.

In experiments with students, the experimental work is usually carried out within a course being followed by the students. Students often have rewards of an academic nature, such as part of the course grade, or extra credits for the student's degree, besides the didactic objectives that the participation on the experiment should have (e.g. the experiment participation involves the practical usage of concepts acquired during the course).

Data collection

The process of data collection corresponds to the actual enactment of the experiment. Experimenters should record information such as the schedule and effort used in the experiment by participants, so that this information can confronted with what was previously planned. Any problems detected on the experiment package should also be registered, so that it can be improved in further replications of the experiment.

Special events concerning the experiment, such as subject's mortality (subjects that are removed from the experiment - in the case of human participants, this happens when a prospective participant ends up not participating) must be recorded for further analysis. Subjects' mortality is important, at least from two perspectives: on the one hand, understanding the motives that lead to mortality of subjects may help improving the experimental design in future replications of the experiment; on the other hand, the potential impact of mortality on the experiments results should also be assessed. Patterns on the mortality of subjects may help uncovering factors which are important to the experiment but are not addressed by the followed experimental design. These factors should be considered, when analyzing the threats to the validity of the experiment.

Data validation

This process aims at ensuring that the experiment data has been collected correctly. Problems with data collection can result from, for instance, erroneous performance (or usage) of collection tools, misinterpretation of data collection forms by the participants in the experiment, or deviations from the planned experimental protocol. The data quality is essential so that adequate inferences can be made from data. This validation process may involve not only the researchers conducting the experiment, but also the participants. The latter can help clarifying data that is found likely to be incomplete, or incorrect.

Problem reporting

All deviations from the original plan should be recorded. Detailed as an experiment plan may be, there are details that may not have been considered while planning, or were insufficiently dealt with at that phase. Identifying those problems and how they were dealt with by the experimenters is an enabling condition for experimental replicability, as well as an important step toward identifying potential threats to the validity of the experiment.

3.3.4 Data analysis

Once the data has been collected, its analysis can begin. This process involves, in essence, three steps: the description of the data set, its reduction, and the testing of the hypotheses defined during the experiment plan (figure 3.17).

Figure 3.17: Experiment data analysis

Data description

The data collected in the experiment should be analyzed by using, in a first moment, adequate descriptive statistics. The set of adequate statistics varies with respect to the target variables under scrutiny. For continuous variables, the count of observations, mean value, median, mode, minimum, maximum, and standard deviation are often collected. For discrete variables, a frequency analysis is adequate. Data description helps in understanding its central tendency and dispersion. If the sample is split into several groups, to accommodate the chosen experimental design, the data description should be performed for each of those groups individually.

Data set reduction

The data description allows detecting atypical cases, such as incorrect, outlier, or extreme values. The cause for atypical cases should be identified and investigated further, since this may help improving the collection process, in the case of incorrect data, or provide more insight into the variables, in the case of outliers and extreme values. A single case that is significantly different from the other cases can bias the subsequent analysis performed on the data set. To avoid biasing the subsequent analysis performed on the data set, the removal of atypical cases should be considered before hypotheses testing.

The most well-known kind of unusual case in a sample is the **outlier**. In a normal distribution, 95% of the cases are less than one standard deviation away from the mean of the sample. A value is said to be a mild outlier if is more than $1.5 \times interquartile\ range$ away from the mean, and an **extreme** outlier if it is more than $3 \times interquartile\ range$ away from the mean.

To understand how outliers and extremes can bias analysis, consider the example of regression analysis. In the context of regression analysis, we can compute the **leverage** of an independent variable as a measure of how far it deviates from the variable's mean. The leverage can be expressed in terms of standard deviations. We can think of leverage analysis as the *"outlier detection"* for the explanatory variables of the regression model.

The **influence** of a case is a measure of the effect of removing that case from the sample, with respect to the the model being computed in the data analysis. In a regression analysis, it measures the extent to which that case influences the regression coefficients. Influence detection can be viewed as a combination of outlier detection with leverage analysis.

In regression analysis, the most commonly used method is the least squares method, which is vulnerable to the presence of outliers, as well as to heteroskedasticity. A sample is said to be heteroskedastic if the variance the independent variable is not constant for all values of the dependent variable[9]. The least-squares method's vulnerability leads to the need for considering removing the most influential cases before proceeding with the analysis. An alternative would be to use a robust regression model, such as MM-estimation [Yohai 87], which addresses these problems, but is not currently supported by some of the most popular statistics packages.

Hypothesis testing

This activity consists in performing the statistical tests that will assess the hypotheses defined in the experiment plan. This involves checking that the pre-conditions for the

[9] A compelling example from social sciences is a regression model using income as the independent variable and expenditure as the dependent variable: the variance of expenditure is typically higher for people with a high income and lower for people with a low income

tests that will be performed are met. Sometimes, restrictions imposed by the data obtained during the experiment may induce adjustments in the hypotheses being tested. If so, such changes should be clearly documented.

Interesting discussions on hypotheses evaluation in the context of software engineering can be found in [Wohlin 99, Singer 99, Kitchenham 02]. They all convey the notion that the soundness of the hypothesis test has to be as verifiable as possible by external observers reading the experiment's report. This is only possible if a detailed description of the tests results, their probability, degrees of freedom, direction, and test power is reported. When trying to combine results from tests performed in independent experiments, this level of detail is essential so that such comparisons are meaningful. A detailed presentation of results is also cornerstone for supporting the results interpretation.

3.3.5 Results packaging

Once the experimental work *per se* is finished, it is essential to package the results so that they can be used either within the context of the organization that sponsors the experiment, or by the community. This involves documenting the whole experimental process, as discussed in the previous sections, and including a discussion on the results achieved with the experiment. This discussion should focus on aspects such as the interpretation of the results, the limitations of the study, the inferencing that can be made with respect to the extent to which the study's results are expected to hold in the population, and the identification of the learned lessons (figure 3.18).

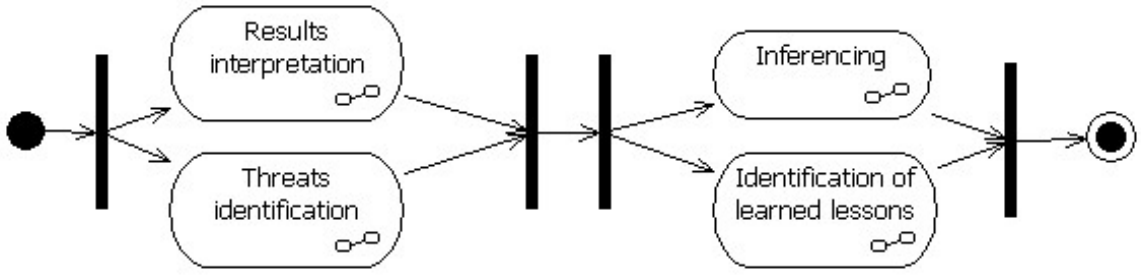


Figure 3.18: Experiment results packaging activity

Results intepretation

This activity concerns the analysis of the outcome of the tests, anchored on the theory that is being assessed through them. When the tests do not confirm the theoretical assumptions, identifying the causes that lead to that failure can be used as a stepping stone to allow the refinement of the theory, or even the construction of a new theoretical framework to explain the object or process under test.

Threats identification

Throughout the experimental process description, we briefly outlined the impact that decisions concerning activities such as sampling, or experimental design selection have on the validity of the results obtained in the experimental process. While packaging the experimental results, one should look back to the whole process and clearly identify the potential threats to the validity of those results. Furthermore, one should discuss the measures that were in place to address each of those threats.

The objective of threats identification is to document potential weak spots of the experimental work being described. Rather than a depreciation of the value of the experimental work, this analysis can be viewed as an active and systematic approach to identifying opportunities for further complementary studies that, as a family of related studies, can contribute to the Software Engineering body of knowledge.

In [Wohlin 99], Wohlin *et al.* identify four kinds of threats to validity and discuss how these threats can be dealt with:

- **Internal validity** is concerned with the validity of the study itself, with respect to the causal effect being studied.
- **External validity** refers to the experimenter's ability to generalize the results from the experiment to industrial practice.
- **Construct validity** concerns the generalization of the results of the experiment to the theory behind it.
- **Conclusion validity** is related to our ability to draw the correct conclusion about the relations between our treatment and the experiment's outcome.

In what concerns the internal validity of the study, we consider two sorts of validity threats: **single group threats**, **multiple groups threats**, and **social threats**. Single group threats can occur from not having a control group in the experiment.

- **History**. This threat concerns uncontrolled events that are irrelevant for the theory being tested, but may nevertheless introduce a confounding effect on the outcome of tests performed after they occur. For instance, consider the bias that can be introduced in an experiment concerning the productivity of code developers, following an event that might break their concentration on the coding task, such as a very exciting sports event, or breaking news on a catastrophe.
- **Maturation**. Occurs when subjects react differently as time progresses. Depending on the type of maturation, the bias may be a positive or a negative one. A positive bias may result of a learning process, for instance. A negative one may result from the saturation of subjects.
- **Testing**. If a test is repeated several times, a non-intentional side-effect, such as learning, can be introduced in the experiment.

- **Instrumentation**. If the measurement instruments are not working as precisely as they should, errors in the measurements may occur, either systematically or randomly. In both events, they may jeopardize the quality of the data collected in the experiment, and, as a consequence, the interpretation of the results. Another possible instrumentation threat comes from changing the measurement instruments during the experiment (e.g. changing a the software metrics collection tool).

- **Statistical regression**. This threat can occur when the subjects of a study are selected for obtaining an extremely high, or an extremely low results in a previous test. When tested again, both are likely to obtain a result closer to the population mean than in the previous test.

- **Selection**. When sampling from a population, there is a risk that the subjects are not representative of the whole population (see discussion on subjects' selection, in section 3.3.2).

- **Mortality**. When the subjects dropping out of an experiment are representative of the population, or one of its subgroups, this has an effect on the overall conclusions that can be drawn from the experiment.

- **Ambiguity about direction of causal influence**. The fact that two variables are highly correlated does not imply that one of them has a direct influence on the other. They can be both influenced by a third variable. When planning an experiment, effort must be put concerning the correct identification of causes and effects.

Multiple groups threats are the result of the multiple groups being exposed differently to single group threats. This may reduce the comparability of results obtained in different groups.

Social threats to internal validity can stem from the usage of differentiated treatments within our sample, if that differentiation causes a change in the behavior of the subjects:

- **Diffusion or imitation of treatments**. If the members of the control group imitate the behavior of the group being tested, this can introduce a bias from the smaller differentiation among groups, contrary to the expectations.

- **Compensatory equalization of treatments** When different groups receive different treatments, the control group may receive some form of compensation from not using the treatment being tested. If that compensation has an effect on the performance of the control group in the experiment, this may jeopardize the conclusions of the overall test.

- **Compensatory rivalry** This can occur if subjects from a group not receiving a new treatment feel they are being penalized and work harder than they normally would to counter that effect.

- **Resentful demoralization** This can occur if subjects from a group not receiving a new treatment feel they are being penalized by this situation and become less involved in the experimental work than their counterparts. It is the opposite situation of compensatory rivalry.

External validity refers to one's ability of generalizing results beyond the scope of the experiment. We consider three potential sources of threat:

- **Selection**. This problem occurs if the used sampling does not provide a representative sample of the population. It may hamper the experimenter's ability to generalize the results of the experiment outside the used sample.

- **Setting**. An experiment can be jeopardized by using an unrealistic experimental environment. For instance, if an outdated development environment is used in an experiment concerning a particular aspect of software development, it may be the case that the same experiment, carried out in a modern development environment would yield completely different results, assuming the tasks being tested have a more sophisticated support in the modern development environment.

- **History**. Refer to the discussion on history threats as single operation threats, earlier in this section. With respect to external validity, a special event biasing the results may damage our ability to extrapolate from them to the most general situation, where that event is irrelevant.

Construct validity threats can assume one of two forms: social and design threats. Social threats result from problems related to the behavior of the subjects and experimenters, if they somehow act differently than they would otherwise, due to the experiment. This behavioral change is a product of the subjects' and experimenters' awareness to the experiment, although it may be unintentional:

- **Hypothesis guessing**. As subjects are aware of being observed in the context of an experiment, they may behave differently to provide a specific impression on the observers. This sometimes leads to hypothesis guessing, where subjects try to figure out what is the hypothesis under study, so that they can perform in the test according to their preferences concerning the hypothesis they think is being assessed.

- **Evaluation apprehension**. If subjects are not comfortable with being assessed in the context of an experiment, a frequent human trait, they may try to provoke a good impression on the experimenters, thus changing their normal behavior.

- **Experimenter's expectancies**. Conversely, the experimenter usually has an interest in the outcome of an experiment. This may bias the conduction of the experiment toward confirming the theory underlying the experiment, or refuting it.

Construct validity design threats result from difficulties in the rigorous definition of the causes and effects being tested. Such difficulties may lead to a poor choice of measurements and treatments, as the theoretical concepts under test are poorly understood. For instance, a subjective concept such as design quality is open to several conflicting definitions. Construct validity design threats include:

- **Inadequate preoperational explication of constructs**. This threat occurs when the constructs involved in an experiment are poorly defined. If the theory underpinning the experiment is not clear, analyzing experimental results becomes more difficult.

- **Mono-operation bias**. Considering a single independent variable, cause, or treatment in a study may introduce the mono-operation bias, because a single independent variable (or cause, or treatment) is always flawed with respect to the construct upon which it is based. The countermeasure is, of course, is to use multiple independent variables (or causes, or treatments, respectively).

- **Mono-method bias**. Using a single kind of measure, or observation, is a threat, in the sense that it may introduce a bias. Using several alternative measures or observations, one can minimize the effect of such bias, by focusing on the commonalities observed with the alternative measurements and observations of the same concept.

- **Confounding constructs and level of constructs**. Sometimes, rather than assessing a construct with respect to its presence, one should focus on its level. Consider the example where the performance of the participants on a code inspection experiment are classified as having experience with a given programming language, or not. Concerning the participants who do have experience with the language, different levels of expertise with it may have a stronger relation with the observed effect than a simple binary assessment of such previous experience.

- **Interaction of different treatments**. When subjects are administered different treatments, it is possible that those treatments interact. That interaction may bias the results of each treatment. It is useful to understand the combined effect of several treatments, to avoid using combinations of treatments that cancel out their benefits. Failing to consider their interaction may lead to erroneous conclusions with respect to each treatment's effect. In short, when several treatments are involved, it may not be possible to distinguish which effects are attributable to each of the treatments and which are a result of the combination of all treatments.

- **Interaction of testing and treatment**. A fundamental part of testing is the application of treatments. Subjects undergoing testing activities may act differently from how they normally would, thus biasing the outcome of the tests. In the context of Software Engineering, this threat is stronger in experiments involving human participants, as we have seen while discussing the social threats to construct validity, earlier in this section.

- **Restricted generalizability across constructs**. Although a treatment may have the desired effect on a construct we are concerned with, it may also have undesired side effects on other constructs that should also be relevant in the analysis of the outcome of the treatment. If side effects on those other constructs are not monitored, there is a risk of drawing conclusions that are not generalizable those constructs. Consider, for example, the introduction of a new component technology that helps improving development productivity (the monitored construct) but also leads to a lower maintainability of the code (the other construct that should have been monitored, but was not).

Conclusion validity threats are threats that are inherent to the usage of statistical tests:

- **Statistical power**. When sample sizes are too small, the value of α is low, or an inadequate statistical test is chosen, a type II error can occur, due to the lack of statistical power. Conversely, a type I error can occur if α is set too high.

- **Violated assumptions of statistical tests**. Each statistical test prescribes a set of pre-conditions that are to be verified before using the test. Failure to comply with those assumptions can endanger the validity of conclusions drawn from such tests.

- **Fishing and the error rate**. When too many tests are performed, there is a chance that some of them will reveal spurious relations between variables, purely by chance.

- **Reliability of measures**. If measures have a low reliability (e.g., they are not stable), they can contribute to an inflation of the error terms, introducing noise in the statistical test.

- **Reliability of treatment implementation**. A lack of standardization in the treatment implementation may lead to a confounding factor, if the treatment is inconsistently administered in different groups. These variations are more likely to happen when different people administer the treatment, although they can also occur with the same person.

- **Random irrelevances in experimental setting**. The experimental setting may contain features that interfere with the outcome of the tests being performed, by

providing sources of variation which are not relevant for the tests. These sources of variation will increase the error variance.

- **Random heterogeneity of subjects**. Heterogeneity of subjects can increase the error variance, as subjects may react differently to treatments.

Inferencing

Following the testing phase of the experimental work, conclusions should follow the statistical tests results, through inference. Considering all the threats previously identified, the researchers have to estimate how the results obtained in the experiment are expected to hold beyond the experiment's sample (i.e. in the population).

Identification of learned lessons

During the whole experimental process, the practical details of conducting that process, including potential gaps or impractical design decisions in the experimental protocol should be registered, along with the approach followed to circumvent these problems. These informations are particularly valuable to other researchers and practitioners who wish to replicate the experiment, as they mitigate to some extent the tacit knowledge problem.

3.3.6 An overview of all the sub-processes

To wrap up our presentation of the ESE process, we now present an expanded version of figure 3.2, in figure 3.19. The top-level activities (requirements definition, design planning, data collection, data analysis, and results packaging) include the information conveyed by the activity diagrams presented throughout section 3.3.

3.4 The experimental process case study

3.4.1 Motivation

One of the crosscutting concerns throughout this dissertation is the praise for the benefits that the experimental validation of claims made in the context of Software Engineering proposals. To provide a preliminary validation of the proposed process model, we next report a case study conducted at Universidade Nova de Lisboa (UNL) with graduate students pursuing a MSc degree in Informatics.

Problem statement

The process model described in this chapter was designed to conform with the state of the art practice in Experimental Software Engineering, while being accessible to

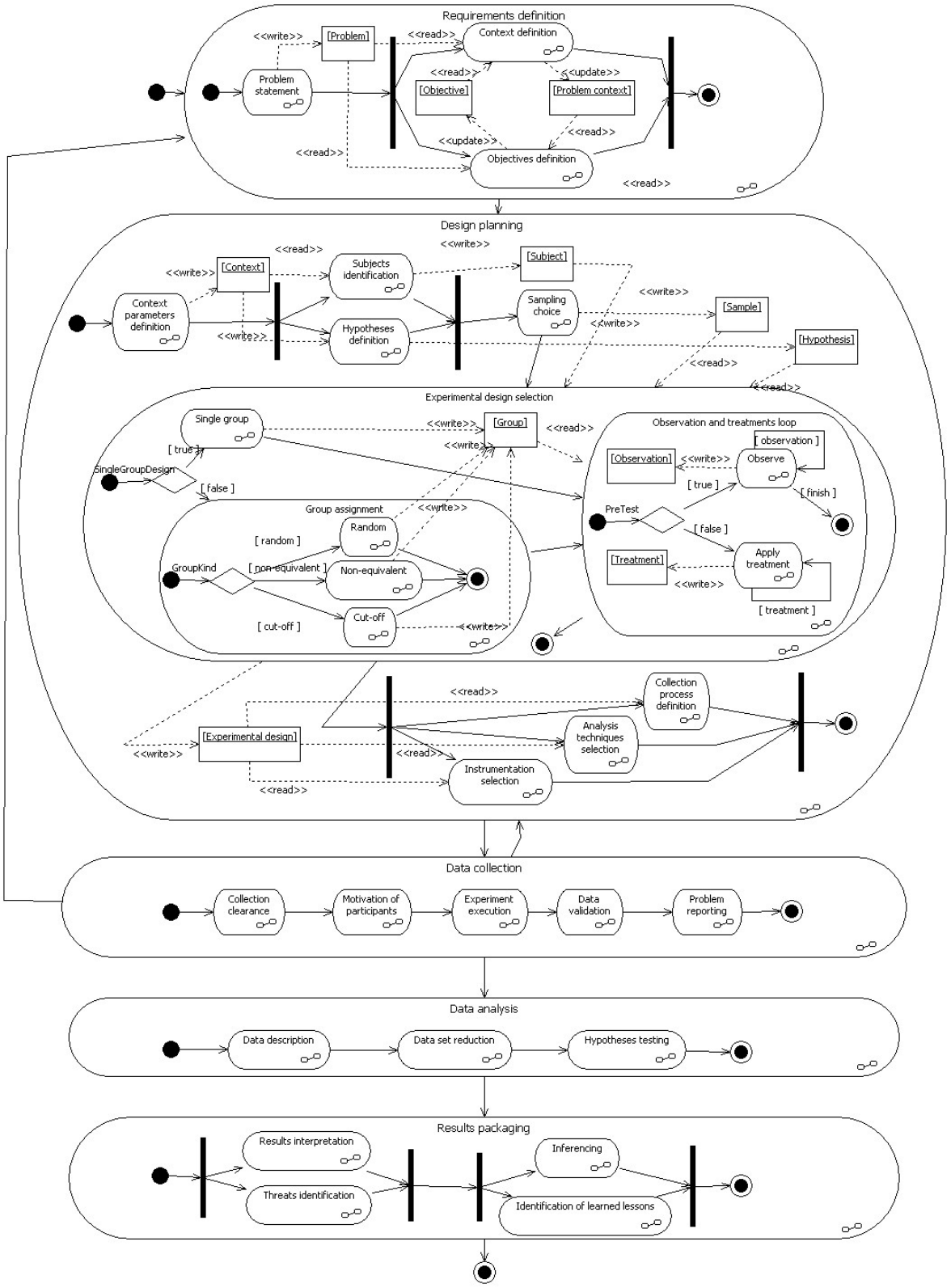


Figure 3.19: Experimental Software Engineering process model

practitioners engaged in experimentation. Although we have been using this process in our work, as will be illustrated throughout the whole dissertation, we would like to assess how hard it is for new experimenters to follow this process.

Among several other desirable quality attributes in a process model such as the one presented in this chapter (e.g. effectiveness, predictability) we would like it to have a balanced learnability and understandability.

In this case study, we analyze the results on a series of experiments conducted by graduate students, following the process model presented in this chapter, in order to identify the sub-processes within our model which constitute a harder challenge for practitioners. This information can be used to improve the presented process model, in the future.

Research objectives

In this case study, our goal (**G1**) is to:

analyze software engineering experiments,
for the purpose of their evaluation,
with respect to the quality of their outcome,
from the viewpoint of a course instructor,
in the context of experimental work carried out during a course for graduate students on the quality of software products and processes.

Context

This case study was conducted in the context of the *Product and Process Quality* course[10], carried out in the Spring semester of 2008 in the context of the MSc program in Informatics, at UNL. This is a 6 ECTS[11] course where, students were asked to conduct an ESE project, following the process presented in this chapter, as part of their evaluation during the semester.

3.4.2 Related work

To the best of our knowledge, this was the first application of the process presented in this chapter in experimental work which was performed by subjects who were not members of the proponent's research group. Other examples of the application of the proposed experimental process can be found throughout this dissertation. A more thorough discussion on related work concerning the model under scrutiny can be found in section 3.5.

[10]The details concerning this course, including course materials are publicly available at `http://moodle.fct.unl.pt/course/view.php?id=1713`

[11]European Credit Transfer System

3.4.3 Experimental planning

In this case study, we will try to achieve goal **G1**, described earlier in this section, by following the plan presented in this sub-section.

Experimental units, material and tasks

As stated in the previous section, this case study was carried out in the context of the 2008 *Product and Process Quality* course at UNL. The participants were graduate students who chose to take this elective course. Typically, these students are in the second semester of Bologna's second cycle, that is, a year away from finishing their MSc degree. The students were grouped in teams of two members. The students had access to the course materials, publicly available on the course's web site, which include not only an earlier description [Goulão 07a] of the process presented here, but also the course's slides, as well as references to several relevant publications about Experimental Software Engineering. Furthermore, the students had access to the **I**nternational **S**oftware **B**enchmarking **S**tandards **G**roup (**ISBSG**)[12] repository. A description of this repository can be found in [ISBSG 07a, ISBSG 07b]. Each group was asked to perform a particular observational study using data stored in ISBSG's repository. This task was performed as part of the evaluation process of the course, off-line.

Hypotheses

The available time schedule and resources conditioned the extent to which this form of validation could be performed. In the best interest of the students taking the course, it was not feasible, for instance, to create a control group that would, for instance, perform similar experiments in an ad-hoc way, so that we could compare the outcomes and use them to assess the benefits of following this process, as opposed to not following it. This constrains the kind of hypotheses we can validate here, as all students followed the same process. We can only discuss qualitatively whether or not the process helped students to successfully completing their tasks (and will do so, in the discussion of this case study).

We can, however, test whether or not the process description made available to the students [Goulão 07a], along with the course training and course materials, lead to a well balanced outcome of the experimental processes. In other words, were students able to follow the process with a consistent degree of success in each of the sub-tasks, or were there sub-tasks that would clearly benefit from improvements in the documentation and training made available to students?

More formally, we can express this concern as hypothesis $H1$, which will be sub-divided into the null hypothesis ($H1_0$) and its alternative ($H1_1$):

[12] http://www.isbsg.org/

$H1_0$: The process was followed with a relatively uniform success.
$H1_1$: The process was followed with significantly (and consistently) different levels of success in different tasks.

Independent variables

The independent variable is nominal and represents group membership. It corresponds to the *group's id* (*GroupID*).

Dependent variables

The dependent variables of these study are the detailed classifications of each group. The specific weight given by the course tutor to each of this partial classifications is not relevant for our analysis. Therefore, we will represent the grades as a percentage of the achieved success. The overall grade of the group in this project is a weighted sum of these partial grades, but its value is not relevant for the hypothesis being tested. The considered dependent variables are represented in the following list by a (**code**), followed by a *short description*. All their values are represented as a percentage:

- (**W1.1**) *Problem statement*
- (**W1.2**) *Context definition*
- (**W1.3**) *Objectives definition*
- (**W2.1**) *Context parameters*
- (**W2.2**) *Hypothesis formulation*
- (**W2.3**) *Variables selection*
- (**W2.4**) *Subjects selection*
- (**W2.5**) *Experiment design*
- (**W2.6**) *Collection process*
- (**W2.7**) *Analysis techniques*
- (**W2.8**) *Instrumentation*
- (**W3.1**) *Collection clearance*
- (**W3.2**) *Motivation of participants*
- (**W3.3**) *Data collection*
- (**W3.4**) *Data validation*

- (**W3.5**) *Problem reporting*
- (**W4.1**) *Data description*
- (**W4.2**) *Data set reduction*
- (**W4.3**) *Hypothesis testing*
- (**W5.1**) *Results interpretation*
- (**W5.2**) *Validity threats identification*
- (**W5.3**) *Inference (generalization)*
- (**W5.4**) *Learned lessons*

Design

This case study can be described as a **within groups, post-test only design**. In Trochim's notation [Trochim 06], this can be described as follows:

```
X O11
X O12
...
X O53
X O54
```

In other words, each subject in our group receives exactly the same treatment, and its performance is then observed with each of the dependent variables (denoted as `Oij`; for instance, `O41` stands for Data description). The rationale is to look for significant differences among the observations (which can be considered simultaneous) that are consistently observed in our subjects.

Procedure

The groups carry out their project in two phases. First, they file in an early version of their project report, after 4 weeks. This report is used for an early control with respect to who is really following the course and acts as a milestone that students have to overcome, in order to successfully complete the course. That said, the deliverable presented at this point is not addressed in this observation. Then, 6 weeks after the early version, they deliver their final project report. Only the latter is evaluated, by granting grades corresponding to each of our dependent variables.

Analysis procedure

The data analysis presented here follows the following steps:

- Descriptive statistics: the mean, standard deviation, minimum and maximum values of all the variables are presented and discussed.
- Data set reduction: if necessary, outliers and extreme values are removed from the analysis.
- Normality tests: these tests are crucial for deciding which are the adequate statistics for our hypothesis, given the characteristics of the distribution in our sample.
- Hypothesis test: Depending on the sample's distribution, a parametric (for normal distribution) or a non-parametric (for other distributions) test is performed to check for statistically significant differences among our observations.

3.4.4 Execution

Sample

14 out of the 17 groups that signed up for the experiment finished the task. Therefore, the mortality of subjects (considering the groups as subjects) is of 17,6%.

Preparation

Before and during the conduction of their experimental work, the participants received training on the several tasks they were to perform in their project.

Data collection performed

The students performed their experiments as part of their normal work within the course. This project accounted for 30% of their final grade, so the incentive to perform well in it was considerable. The validation effort of our proposal did not interfere directly in the outcome of their projects, as this case study's data is based on the evaluation of their reports. From the participant's point of view, this was a normal project in a course. The classification of the experiment reports was carried out by the course instructor[13]. The data analysis that follows was based on the detailed classification report we had access to.

[13]The course instructor was Prof. Fernando Brito e Abreu, the supervisor of this dissertation's proponent.

3.4.5 Analysis

Descriptive statistics

Table 3.1 presents the descriptive statistics for the collected variables.

	Mean	Std. Deviation	Minimum	Maximum
W1.1	,6250	,16261	,50	1,00
W1.2	,6964	,20045	,25	1,00
W1.3	,7500	,24019	,50	1,00
W2.1	,7500	,21926	,50	1,00
W2.2	,7857	,21611	,50	1,00
W2.3	,6786	,20636	,25	1,00
W2.4	,6250	,27298	,25	1,00
W2.5	,6786	,28468	,00	1,00
W2.6	,4821	,22922	,25	1,00
W2.7	,6250	,32150	,00	1,00
W2.8	,6250	,25476	,00	1,00
W3.1	,7679	,26790	,00	1,00
W3.2	,6250	,33613	,00	1,00
W3.3	,5179	,26790	,00	,75
W3.4	,3750	,25476	,00	,75
W3.5	,3036	,29708	,00	1,00
W4.1	,6964	,24374	,25	1,00
W4.2	,6071	,21291	,25	1,00
W4.3	,6964	,24374	,25	1,00
W5.1	,6607	,23220	,25	1,00
W5.2	,5000	,24019	,00	,75
W5.3	,6071	,30562	,00	1,00
W5.4	,6250	,25476	,00	1,00

Table 3.1: Descriptive statistics

Table 3.2 presents the normality tests for our dependent variables. The null hypothesis for the normality tests (the Kolmogorov-Smirnov and the Shapiro-Wilk tests) is that there is no statistically significant difference between the observed accumulated distribution and the one of the theoretical distribution being tested (the normal one). Several of the variables have a non-normal distribution according to at least one of the tests. Considering a confidence interval of 95% in both tests, the normality hypothesis should be rejected if the significance of the test is less than *0,05*. In other words, if the variable's normality test has a significance level (p-value) greater than *0,05*, we can assume the variables' distribution to be normal, with a confidence level of *95%*. The non-normal variables (according to at least one of the normality tests) are highlighted in bold, in table 3.2, as is the test significance that points to the data's non-normality.

Data set reduction

No data reduction was performed, at this point.

	Kolmogorov-Smirnov(a)			Shapiro-Wilk		
	Statistic	df	Sig.	Statistic	df	Sig.
W1.1	,350	14	***,000***	,731	14	***,001***
W1.2	,320	14	***,000***	,850	14	**,022**
W1.3	,280	14	***,004***	,730	14	***,001***
W2.1	,230	14	**,043**	,792	14	***,004***
W2.2	,268	14	***,007***	,786	14	***,003***
W2.3	,421	14	***,000***	,697	14	***,000***
W2.4	,176	14	,200(*)	,888	14	,075
W2.5	,313	14	***,001***	,842	14	**,017**
W2.6	,255	14	**,014**	,843	14	**,018**
W2.7	,164	14	,200(*)	,906	14	,140
W2.8	,331	14	***,000***	,814	14	***,007***
W3.1	,331	14	***,000***	,736	14	***,001***
W3.2	,225	14	,053	,867	14	**,038**
W3.3	,259	14	**,012**	,792	14	***,004***
W3.4	,260	14	**,011**	,876	14	,052
W3.5	,204	14	,119	,844	14	**,019**
W4.1	,218	14	,069	,875	14	**,049**
W4.2	,249	14	**,019**	,883	14	,065
W4.3	,218	14	,069	,875	14	**,049**
W5.1	,256	14	**,014**	,874	14	**,049**
W5.2	,214	14	,081	,861	14	**,032**
W5.3	,180	14	,200(*)	,923	14	,241
W5.4	,331	14	***,000***	,814	14	***,007***

Table 3.2: Normality tests for the dependent variables. The values marked with (*) are lower bounds for the true significance of the Kolmogorov-Smirnov test. (a) stands for Lilliefors significance correction. We cannot assume a normal distribution of the variables in **bold**. The significance of tests is highlighted in **bold** for tests with $p < 0,05$ and ***italic bold*** for tests with $p < 0,01$.

Hypotheses testing

As we have seen, the data does not have a normal distribution. As such, we have to use non-parametric tests. The non-parametric tests that we will perform to validate hypothesis $H1$ rely on the ranks of the values in the sample, rather than on the values themselves. The rationale for using ranks is to avoid the assumption of normality in analysis of variance. All the grades (ranging from W1.1 to W5.4) are put into a large sample, and ranked, for each group, from the lowest to the highest value. Table 3.3 presents the mean rank, for each of the variables, considering all groups.

Sub-process	W1.1	W1.2	W1.3				
Mean Rank	11,6	13,6	15,2				
Sub-process	W2.1	W2.2	W2.3	W2.4	W2.5	W2.6	W2.7	W2.8
Mean Rank	15,4	15,8	13,4	12,0	13,8	8,0	12,3	12,4
Sub-process	W3.1	W3.2	W3.3	W3.4	W3.5			
Mean Rank	16,2	12,3	9,9	6,0	4,6			
Sub-process	W4.1	W4.2	W4.3					
Mean Rank	13,5	11,4	13,1					
Sub-process	W5.1	W5.2	W5.3	W5.4				
Mean Rank	12,5	8,9	12,2	12,0				

Table 3.3: Ranks of the grades, for testing hypothesis **H1**.

The problem, then, is to find out whether or not any of these mean ranks differs significantly from the remaining ones. We will use two non-parametric tests to do so.

Table 3.4 presents the results of the Friedman test, a non-parametric test designed to detect differences in treatments across multiple tests attempts [Friedman 37]. This test is commonly used as a non-parametric alternative to the Analysis of Variance test. Recall that our null hypotheses states that "the process was followed with a relatively uniform success". There is a significant difference among the results of the treatments, with a chi-square of 63,980,(22, N=14), and p=,000<,01. Therefore, we can reject the null hypothesis[14]. In other words, at least one of the sub-processes lead to an outcome significantly different from the remaining ones.

N	14
Chi-Square	63,980
df	22
Asymp. Sig.	,000

Table 3.4: Friedman test for hypothesis **H1**.

We can further explore this by using Kendall's W test, which is a normalization of Friedman's test and is used to assess the level of concordance between raters. A strong agreement is signaled by a Kendall statistic value close to one, while a strong disagreement presents a value close to 0. The 0,208 value in table 3.5 indicates a low but significant agreement level with a chi-square of 63,980,(22, N=14), and p=,000<,01.

N	14
Kendall's W(a)	,208
Chi-Square	63,980
df	22
Asymp. Sig.	,000

Table 3.5: Kendall's W test for hypothesis **H1**. (a) stands for Kendall's Coefficient of Concordance.

The most likely candidates for the existing agreement and, likewise, for the significant differences found while evaluating the reports, can be identified using a boxplot representation of the distribution of the average grades for each of the sub-processes (left side of figure 3.20). Sub-processes W3.5 and W3.4 have an extreme and an outlier mean classification. If we remove these sub-processes, and remake Friedman and Kendall's tests, the statistics are still significant (p = 0,034 < 0,05) (left side of table 3.6). The new boxplot reveals that, in the absence of W3.5 and W3.4, three sub-processes emerge as outliers (W2.6, W5.2, and W3.3). If we remove W2.6 from the sample (the one which is further away from the mean value), both Friedman's and Kendall's tests

[14]The "traditional" way of interpreting a chi-square test is to use the *chi-square table*. If the calculated chi-square value is greater than the critical value in the table, for a given significance and number of degrees of freedom, we can reject the null hypothesis. However, modern statistics tools, such as SPSS, compute the significance level directly, to save users the burden of consulting those tables. The asymptotic significance presented by the used statistics tool has a value lower than 0,01 (in fact, lower than 0,0005), we can reject the null hypothesis.

no longer report statistically significant differences between the different assessments of the sub-processes under scrutiny (right side of table 3.6).

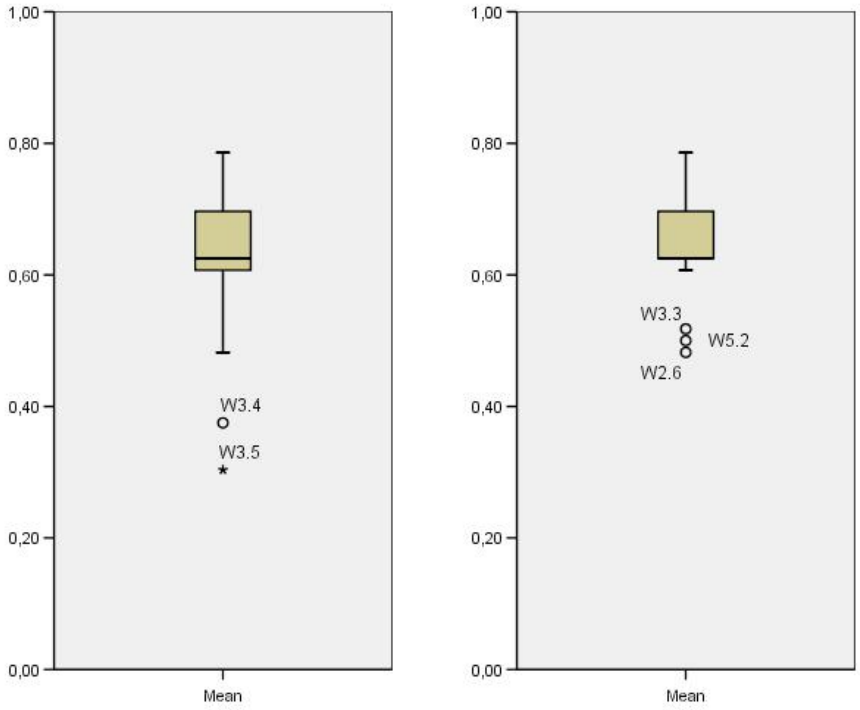


Figure 3.20: The boxplot on the left presents the distribution of the classifications, including all the sub-processes. Sub-process W3.5 is marked as an extreme. Sub-process W3.4 is marked as an outlier. The boxplot on the right side presents the distribution of the classifications, if we exclude the extreme W3.5 and outlier W3.4. Note that, in the absence of these two sub-processes, three other sub-processes (W2.6, W5.2, and W3.3) are now considered outliers in the remaining sample.

3.4.6 Interpretation

In an ideal process, practitioners should be able to carry out all the sub-processes with a consistently high proficiency. The Friedman test, complemented by Kendall's coefficient of concordance lead us to think that there are some parts of the process that were handled with a significantly different success by the participants in this case study, when compared to the others. The fairly low concordance coefficient also points to the fact that students roughly achieved the same success level in the majority of the sub-processes. In an ideal process, Kendall's coefficient of concordance should be close to 0. This would mean that we were not able to significantly rank the success of the different sub-processes. Of course, the evaluations should also be high, as there is not much point in having practitioners performing consistently bad in all sub-processes.

While for most of the process the results are quite encouraging and balanced, considering the lack of experience of the subjects in conducting experimental work, we should check what happened with the sub-processes where the success was significantly different (and, in this case, lower).

Sample, without W3.5 and W3.4	
Friedman Test Statistics	
N	14
Chi-Square	32,972
df	20
Asymp. Sig.	,034

Sample without W3.5, W3.4 and W2.6	
Friedman Test Statistics	
N	14
Chi-Square	24,288
df	19
Asymp. Sig.	,185

Kendall's Coefficient of Concordance	
N	14
Kendall's W(a)	,118
Chi-Square	32,972
df	20
Asymp. Sig.	,034

Kendall's Coefficient of Concordance	
N	14
Kendall's W(a)	,091
Chi-Square	24,288
df	19
Asymp. Sig.	,185

Table 3.6: Friedman test statistics and Kendall's Coefficient of Concordance, when removing the extreme and outliers that caused the statistically significant differences in the classification of the sub-processes.

By using the extremes and outliers detection, we were able to single out the processes responsible for the concordance that does exist. The three identified sub-processes are the ones with the lowest mean classifications. W3.5, W3.4, and W2.6, correspond, respectively, to *problem reporting*, *data validation*, and *collection process*. Why did our subjects perform poorly in these tasks?

There are at least two plausible explanations for this. The first one is that they may have found these sub-processes' descriptions less clear. But it may also be the case that the experimental tasks they were performing had a role to play in these difficulties. Unlike what usually happens in experimental work, the data used in these projects was collected *a priori* in the **ISBSG** repository. The three sub-processes relate to data collection, to some extent, something that our subjects did not perform, in practice.

Problem reporting focuses on deviations from the experiment execution plan. In this case, the data to be used was readily available from a repository, and the participants showed difficulties in critically assessing, based on the existing information, the problems that may have occurred.

Likewise, the data validation was also one of the weakest sub-processes, again due to the challenges detecting, from the information available in the repository, potential problems concerning data validity. We believe these difficulties may have been increased by the fact that our subjects were novice experimenters.

Finally, providing a detailed description of the data collection process was also challenging for our subjects. Again this may result from difficulties in extracting the relevant information from the repository, on the one hand, and in acknowledging those difficulties, on the other. This is also a typical problem with novice experimenters that we have observed in other *fora*, namely while serving as reviewers for program committees in conferences and workshops.

Evaluation of results and implications

The significant differences found in the sample point us to the parts of the process in which the case study participants performed significantly worse. This information can be used for guiding improvements in the process model, as well as in future editions of the course. With respect to the process model, these improvements can be achieved through a clarification of the description of the process. To a certain extent, we have already done so while writing this chapter of the dissertation. The degree of detail provided here is greater than the one used in [Goulão 07a], not only due to less constrained size, which lead to the inclusion of more details in any of these topics, but also as a result of the feedback collected since the publication of the process, both from our peers and the students participating in this case study.

Threats to validity

It was not feasible to create a control group of students who would conduct their projects using some other process than the one described here. As such, it is not possible to directly and soundly compare the outcome of these projects with a control group of other projects following some other approach to their experimental work. From the course's instructor qualitative point of view, the overall quality of the projects was considered a good improvement, compared to projects conducted in earlier instances of the course, leading to a better structured and overall more mature outcome. However, none of these improvements can be directly assessed from the data provided by this case study alone.

The time constraints imposed by the course's schedule lead to the option of using a repository of software projects as the data source for the experimental work used in this case study. Although this has a clear benefit for students, since they were working on real data, rather than toy examples, the price to be paid is that they did not go through the time-consuming task of collecting data themselves. As discussed earlier, that task provides experimenters with a greater sensibility to the challenges created by some of the sub-processes described in this chapter, and this may have played a role in the outcome of this case study.

Another possible threat is the usage of graduate students as subjects in this assessment. We do not regard this as a major threat, because, as we discussed in the beginning of this chapter, the software industry in general is yet to achieve a mature status when it comes to experimentation. We believe the problems faced by our students do not significantly differ from those that would be felt by more senior practitioners, with the possible exception of practitioners from the ESE community. On the other hand, it is possible that the latter's performances would be biased by their own background.

The size of the sample may also be considered a threat, which we expect to mitigate in future replications of this study.

Inferences

In general, the ESE process followed by the subjects in our case study lead to satisfactory results. The overall quality of experimental reports gives us confidence concerning the suitability of the process described in this chapter for training new professionals in the empirical validation of Software Engineering claims. With the exception of a few sub-processes that we have identified and discussed here, subjects were able to follow the process and produce experimental reports packaged in such a way that their comparison and the replicability of the experiments they describe are facilitated.

The fact that our subjects collected the information from a repository of software process data may have been a confounding effect. This is a deviation from a frequent scenario where the experimenters also have to work on the data collection part of the process. However, using data from a repository is also frequent, particularly when assessing legacy systems. Therefore, while we expect this process to be accessible to practitioners who also handle the data collection part of the process, further experimentation would be required to assess data collection more thoroughly.

These results were obtained with graduate students, who are, in our opinion, comparable to novice experimenters. Extrapolating the observed behavior for more seasoned experimenters is risky. However, we were encouraged by the results obtained by our subjects, and it seems likely that those results will also apply to seasoned experimenters, with a positive *nuance*: seasoned experimenters are likely to have developed the skills that may help them mitigating the possible shortcomings of a process model. So, while we would expect an overall performance improvement with seasoned experimenters, this improvement would probably be more noticeable precisely in the sub-processes were novices had their worst performances.

Lessons learned

This case study provided us with valuable information for two different contexts. For our research work, the feedback provided by students while learning and applying the experimental process presented in this chapter helped improving its description for this dissertation. On the other hand the differentiated difficulties that may have been created by the data collection performed on a repository, rather than directly by the subjects, seem to be an extra challenge for course instructors, as they may need to compensate the absence of that experience from the students. If students consistently perform worse in a delimited set of sub-processes, care should be put in improving the way these sub-processes are addressed in future editions of the course.

3.4.7 Case study's conclusions and further work

The first impressions on this process model are encouraging, in the sense that we found it useful not only in our research, but also as a tool that we can provide to other exper-

imenters to facilitate their work. Although this case study has pointed out some difficulties in the adoption of this process by novice experimenters, the overall outcome of the process's usage was satisfactory.

In the future, we would like to expand and diversify this assessment, through replications of this case study. As discussed in this chapter (section 3.2.2), this could be achieved through a series of close and differentiated experiment replications.

An example of a close replication would be to reuse the **ISBSG** repository in future editions of this course and propose a similar set of projects to new sets of graduate students. This would mitigate the threat concerning the sample size.

To deal with the other identified threats, differentiated replications of the assessment would be required. Examples include:

- Repeating the experiment in a similar context, but using different process models would allow assessing the relative merit of our process, using the other processes as a baseline.
- Following the process proposed in this chapter, modified with enhancements to mitigate the difficulties shown by students in this first case study would clarify whether the difficulties felt by the students were inherent to the process or to some other factor, such as practitioner's background, or to the usage of previously collected data.
- In contrast with the previous replica suggestion, we can change the experimental work proposed to our subjects, so that the subjects have to collect the data themselves, as the usage of previously collected data collection was identified as a potentially confounding effect.
- Changing the characteristics of the sample of subjects used in the experiment would alleviate any concerns with respect to the usage of students in this first validation.

3.5 Related work

3.5.1 Experimental Software Engineering process models

The major novelty of our proposal stems from the integration of dynamics modeling to support the discussion of the several activities, and respective deliverables, involved in Experimental Software Engineering practice. By doing so, we merge three basic views into a unified one:

- experiment reporting guidelines (e.g. [Jedlitschka 05b]);
- experiment conduction guidelines (e.g. [Wohlin 99, Kitchenham 02]), which also encompass reporting guidelines;

- models for representing experimental and quality data (e.g. [Kitchenham 01,Garcia 04,Jeusfeld 98]).

Experiment reporting guidelines

Our dynamic process description captures the main activities that are required so that Jedlitschka and Pfahl's reporting guidelines for controlled experiments in Software Engineering [Jedlitschka 05a] can be followed. While the deliverables of each activity can be mapped to their proposal, we adapted our model so that other (weaker, but often more feasible) forms of empirical work, such as observational studies, and quasi-experiments, can also be supported by a process adhering to our model.

Experiment conduction guidelines

Our process specification, and the data model behind it support the concepts conveyed in empirical research guidelines, such as [Wohlin 99, Kitchenham 02]. While more detailed discussions on each of the process steps can be found in those documents, here we favored the integration of all these concepts into a single model, with the added value of also defining a metamodel structure to support data collection throughout the whole process.

Models for representing experimental data

Kitchenham and Hughes proposed a method for specifying models of software data sets that capture the definitions and relationships among software measures [Kitchenham 01]. The rationale is that without proper safeguards, it is very difficult to ensure that data from different sources can be safely combined and analyzed. Software measurements have to be fully defined, rather than just named, to allow for the repeatability of their collection. Kitchenham and Hughes's work, is focused on the storage of the information concerning the metrics's definitions and values (including collected values, both current and collected in the past, estimates and targets), at different levels of granularity. Our ontology-based representation of collected metrics complements this idea with our approach to automate metrics specification and collection, which we will discuss in chapter 4.

Garcia *et al.* defined an ontology for software measurement that represents concepts including the measurement activities, measures definitions, and metrics definitions [Garcia 04]. Our work shares the concern of having such an ontology description upon which we can describe experimental work, but covers a wider spectrum of experimental concepts and adds the element of dynamics description.

Jeusfeld *et al.* proposed a metamodel for modeling quality management issues in the context of data warehouses [Jeusfeld 98]. Their metamodel is inspired by the GQM approach [Basili 94], and covers goal specification, the construction of queries for as-

sessing the goals, and the definition of metrics to support those queries. Again, this work is focused on metrics definition, starting from goal specification, but the emphasis is on collected information alone.

3.5.2 Alternatives to experimental results evaluation

Our experimental process model description focuses on the most typical case with Experimental Software Engineering analysis practices, referred as *classical*, or *frequentist* in [Kitchenham 02]. Other quantitative approaches to analysis can be used.

Fenton, has been deeply engaged in the research of the usage of a Bayesian approach to the analysis of ESE data. A Bayesian network is a directed graph combined with an associated set of probability tables. In the graph, nodes represent variables that may be either continuous, or discrete. The arcs represent possible causal or influential relationships between the nodes. Bayesian networks allow modeling and reasoning about uncertainty, as they represent the assumptions about the impact of the different forms of evidence represented by the nodes into each other (through the arcs). The graph representation also provides traceability, a feature which is useful when using Bayesian networks to assess complex cause-effect relationships.

Bayesian networks have not been used often within the scope of ESE [Kitchenham 02]. A possible explanation for this was the lack of adequate tool support for them, until the mid 90's, according to [Fenton 02].

Nevertheless, some examples of their usage have been put forward recently, particularly concerning software fault proneness. Fenton *et al.* claim that Bayesian networks can replace with advantage simple regression models for estimating software defect proneness, not only for their traceability property, that facilitates model understanding, but also for their improved accuracy [Fenton 06].

Other authors, such as Sahraoui *et al.* have also been engaged in software quality modeling using Bayesian-based techniques [Sahraoui 01]. More generally, other data mining techniques can be applied to Software Engineering repositories although exploring this avenue of research is beyond the scope of this dissertation.

3.5.3 Qualitative approaches to evaluation in Software Engineering

Although this dissertation is focused on quantitative approaches to evaluation in the context of Software Engineering, there are other evaluation approaches which are inherently qualitative. **Feature analysis** [Kitchenham 96a] is a technique where the evaluator identifies the requirements that stakeholders have for a given software product, or process, and compares them to the features offered by candidate products or processes. This is an essentially qualitative technique in the sense that it requires a subjective assessment of the relative importance of different features and of how well they are implemented in the scrutinized products, or processes. There are several variations

on qualitative approaches that can be used. These include [Kitchenham 96a]:

- **qualitative screenings** - a single individual chooses the features, rating scales, and performs the analysis

- **qualitative case study** - an evaluation performed by an individual who has used a tool, or followed a process, in a real-world environment.

- **qualitative experiment** - a group of individuals make the evaluations, typically after performing a set of typical tasks with the product under scrutiny (or following the process, in a process assessment)

- **qualitative survey** - similar to the qualitative experiment, but the participation in the survey is at the discretion of the potential participant, unlike what happens in the experiment.

- **qualitative effects analysis** - a subjective assessment of the quantitative effect of using a tool, or following a process, based on expert opinion.

In chapter two, we have seen two examples of qualitative screenings: the survey on component models, and the survey on metrics for software components. In both surveys, we identified a fixed set of features to characterize the subjects of our samples (component models and metrics, respectively), and then discussed, for each of those features, the extent to which each of the subjects supported them.

A thorough discussion on the strengths and weaknesses of each of these variations of qualitative approaches is beyond the scope of this dissertation. Nevertheless, we can highlight a few similarities with their quantitative counterparts, with respect to the concerns on their validity.

For instance, while some of these methods rely on the judgment of a single individual (qualitative screenings and case studies) the others can mitigate personal biases that individuals might bring into their assessments.

Another interesting similarity to quantitative studies occurs when we contrast a qualitative experiment and a qualitative survey. The latter has an extra threat to its validity due to the ability of potential participants to choose whether or not they will participate in the survey. It may be the case that people refusing to participate have a common characteristic that is relevant for the assessment they would provide and is therefore biasing the results of the survey. Of course, the same concern applies to people choosing to cooperate in the survey. An extreme example of this threat would be a survey conducted by software company X, comparing two competing components produced by rival companies X and Y, which could be biased by including a large number of respondents from company X, and few from company Y, who could be reluctant to participate in the survey.

3.5.4 Benchmarking

Benchmarking consists in running a predefined set of tests on several competing products, or processes, in order to provide a comparison of their relative performance. This technique lies somewhere between a quantitative and a qualitative approach because, although the data collection carried out during benchmarking has a quantitative nature, the process of developing or selecting the adequate benchmark tools is subjective in nature, and in that sense closer to feature analysis.

Typical examples include the benchmarking of tools for assessing their relative performance with respect to some well-defined criteria (e.g. time and physical memory required to perform a pre-established set of operations). For instance, one can compare different compilers by compiling the same source code and then logging compilation times and detected errors. Benchmarking can also be used in process comparison. For instance, benchmarking has been used for software inspections in [Wohlin 02], where both qualitative benchmarking (e.g. upon the characterization of the various inspection techniques) and quantitative benchmarking (e.g. upon measurable aspects of inspections, such as the percentage of defects found in each inspection) were used.

One of the challenges with benchmarking is how to keep the benchmarking fair, in the sense that, depending on the chosen set of tests, different products may be regarded as the best. It is common for communities to develop benchmarks for their domain-specific targets, therefore creating a common tool for comparative evaluation of products and processes. Another challenge is that benchmarking implies the usage of multiple points of comparison, so that we have a representative sample of products, or processes [Wohlin 02].

3.6 Conclusions

In this chapter we proposed a model for the ESE process that supports the definition of the activities and corresponding deliverables, involved in ESE practice. The process model description of experimental work builds on related work described in section 3.5. The main novelty of this model is the integration, in a homogeneous framework, of the contributions of other researchers, along three threads: (i) experiment conduction guidelines, (ii) experiment reporting guidelines, and (iii) models for representing experimental data.

The proposed process model captures the best practices (in some cases), or state of the art (in others) in ESE and is aligned with a recent proposal for experimental data dissemination (Jedlitschka and Pfahl's reporting guidelines [Jedlitschka 05a]). It can be used as a guideline for practitioners involved in leveraging data collection activities to improve the software process in their organizations, both in industrial and in academic contexts. It may also be used as a framework for supporting experiments' comparison, which was identified as a major need for future Software Engineering research.

Chapter 4

Ontology-driven Measurement

Contents

Background: *Experimental replicability and comparability depend not only on a process such as the one defined in the previous chapter, but also on successfully dealing with the metrics ill-definition problem presented in chapter 2.*

Objectives: *We will present an approach, called Ontolgy-Driven Measurement (ODM) to deal with the metrics ill-definition problem.*

Methods: *ODM combines a domain ontology with metrics defined upon that ontology, using the Object Constraint Language. We illustrate ODM through a case study to independently validate a metrics set for JavaBeans.*

Results: *ODM allows the formal definition of metrics, solving the metrics definition problem, while using a language that is familiar to UML practitioners. The approach also allows defining heuristics.*

Limitations: *ODM requires a suitable metamodel. Such metamodel may not always be available. In those situations, we have to design that metamodel, so that we can use this approach.*

Conclusions: *We successfully formalize and collect metrics using the ODM approach. ODM facilitates the external validation of metrics proposals. Our results with the JavaBeans metrics set point to problems in the external validity of the heuristics for those metrics.*

4.1 Revisiting metrics proposals limitations

In our overview of metrics proposals for CBD, presented in section 2.5, we identified three recurrent problems:

- lack of an underlying context
- metrics ill-definition
- insufficient metrics validation

In this chapter, we will outline our strategy for mitigating these shortcomings and frame it in the context of the Experimental Software Engineering process presented in chapter 3.

4.1.1 Providing adequate context for metrics proposals

As discussed in section 2.5, the lack of an underlying context is a common problem in most metrics definition and makes the interpretation of metrics values troublesome. If we are unable to interpret the values of collected metrics, they become irrelevant for the CBD process.

There is a well-known and widely accepted approach named Goal-Question-Metric [Basili 94] that aims at guiding the definition of software metrics, but the results of our survey, presented in section 2.5 showed that the CBD community is still not using this approach as much as it would be desirable. A quality model for CBD would provide a context for the establishment of goals for which research questions can be made, leading to the definition of objective metrics to support the answer to those questions.

There is no generally accepted quality model for CBD. We can adapt or create a model (or at least identify quality attributes) for providing context to the metrics definitions. The two basic options for a quality model would be to aim for a general purpose quality model for CBD, or to use very specific (and, in that sense, limited in their scope) quality models for some niches within CBD. Our observation of existing CBD quality model proposals (e.g. [Bertoa 02,Simão 03,Bertoa 06]) lead us to prefer the latter option (specific quality models, or small sets of quality characteristics), because it is more feasible to validate such models than to validate general purpose ones.

4.1.2 Toward a sound and usable approach to metrics definition

The metrics ill-definition problem results from either using an at least partially informal approach to metrics definition, making those definitions subjective, or using a formal approach to metrics definition that, although sound, is based on formalisms which are difficult to grasp by common practitioners, making their adoption difficult.

The effort required for large scale manual metrics extraction is prohibitive. Automating metrics collection based on informal definitions is troublesome, because different implementations may be based in different assumptions on the subjective details of those informal definitions. The automation of metrics collection is also a problem when using formal notations. It is often the case that the tools required for using the formal definitions are not supported by major software vendors, but rather by researchers who lack the resources for ensuring a professional tool maintenance and support.

In our opinion, the key for obtaining a metrics specification that is simultaneously formal, understandable, and executable, is to use a formal language which is widely adopted by the software development community, and for which there is tool support available from major integrated development environment (IDE) developers. Ideally, metrics collection should be integrated in such IDEs, so that it can be part of the development process.

4.1.3 Facilitating metrics validation

The Experimental Software Engineering process, described in the previous chapter, is useful in this context, since it was built to frame the set of activities leading to the conduction of experimental work in a sound, comparable, and replicable way. The process was designed to adhere to a common set of research reporting guidelines for presenting the results of experimental work in Software Engineering, proposed in [Jedlitschka 05a].

The support for automated metrics collection is fundamental to foster independent validation efforts. Manual collection of metrics has been shown to be error prone and vulnerable, for instance, to the lack of adherence to sound, widely accepted, coding principles[1] [Counsell 07].

Unfortunately, to the best of our knowledge, most of the proposed metrics referred to in the survey presented in section 2.5 were only tested by their authors, using proprietary or experimental, non-publicly available, tool support, therefore limiting experiments replication. This limits knowledge sharing, both in the research and practitioner's communities, hampering results comparison.

The approach for metrics definition presented in this chapter is aimed at facilitating metrics definition and collection, so that metrics collection is fully replicable, even when performed by different people, using their own tool support for automating the metrics collection process.

[1] Counsell's paper [Counsell 07] refers to metrics collected on the source code, but it is fair to admit that similar problems may occur when manually collecting metrics from other artifacts. Consider, for instance, the difficulty in manually collecting metrics from design documents, where it is common to show only partial views of the design elements that allow focusing on a particular design goal. The information required for computing a particular metric may be scattered throughout several different diagrams with possible overlaps among them.

4.2 Defining Ontology-Driven Measurement

The metrics ill-definition problem can be solved by using a technique proposed by Abreu [Abreu 01b] that we will refer to as **Metamodel-Driven Measurement** (M2DM). M2DM combines the usage of a metamodel of the domain upon which the measurements will be made, with the usage of a formal specification language. The metamodel provides the context information, which is missing (at least in a non-ambiguous way) in metrics specifications where a combination of natural language - to describe the context - and mathematical formulas is used. The formal language allows the specification of the metrics, again in a non-ambiguous way, as the counting rules and their relationship with the underlying domain are hard-coded in the metrics definitions.

M2DM has a limitation: the metrics have to be defined at the metamodel level. Throughout this dissertation, we will use an evolution of M2DM and call it Ontology-Driven Measurement (ODM). This evolution extends the M2DM technique to models which may not necessarily be metamodels. Note that a metamodel is a **model of a model**. In this sense, we can think of ODM as a generalization of the M2DM approach that removes the constraint that metrics can only be defined at the metamodel level. In ODM, metrics can also be defined at the model level. The ontology is a **model of a domain**, but the domain is not necessarily a model. This distinction will become more clear in chapter 6, where we will use metrics defined upon a model that is not a metamodel.

4.2.1 Aligning the approach with a standard

The concept of ODM is generic, both in what concerns its mapping to the chosen ontology specification technology and the formal language that is used to express the specification of the metrics. When defining ontologies (metamodels and models), we will follow an approach aligned with the OMG's current specification technologies. So, we will specify the metamodels and models in UML, and use OCL for defining and collecting metrics.

A layered architecture

The Meta-Object Facility (MOF) is a platform-independent metadata framework used by OMG and considered a cornerstone of OMG's Model-Driven Architecture initiative. It provides the basic building blocks for the specification of metamodels, regardless of these metamodels being object-oriented, or not. MOF 2.0 shares a common core package with the UML 2.0 infrastructure library [OMG 06b], represented as `Core`, in figure 4.1. This `Core` package acts as an architectural kernel for other OMG metamodels, such as the Comon Warehouse Metamodel (CWM) [OMG 03a], the UML 2.0 metamodel [OMG 06b, OMG 05b], the Corba Component Model (CCM) metamodel [OMG 02a]

and the Object Constraint Language (OCL) metamodel [OMG 03b]. The `Core` itself is a complete metamodel.

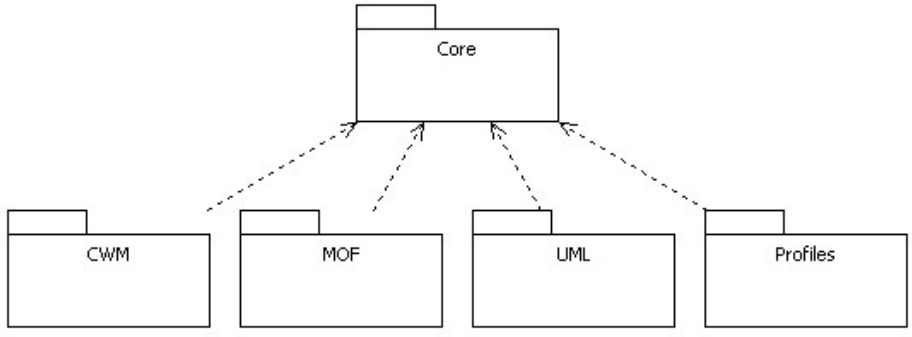


Figure 4.1: OMG's common Core package and its relation to other metamodels

When focusing our attention on the UML 2.0 metamodel, we note the existence of two main parts: the infrastructure and the superstructure. The UML 2.0 infrastructure library [OMG 06b] uses fine-grained packages to bootstrap the the rest of UML 2.0. These packages are useful so that we can build on them to specify other metamodels, using the basic UML 2.0 notation. The main packages in the UML 2.0 infrastructure are the `Core` package, discussed above, and the `Profiles` package, where the constructs used for defining UML extensions are defined (figure 4.2).

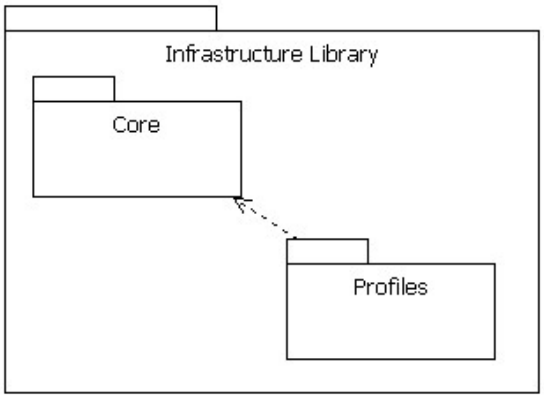


Figure 4.2: UML infrastructure library

Three key concepts to understand metamodeling are those of **classifier**, **instance**, and the **ability to navigate from the instance to its classifier**. A classifier is a classification of instances: it describes a set of instances that have features in common. Each instance has exactly one classifier which describes it. Building on these concepts, it is possible to create a **layered architecture** where the instances at level n have their corresponding classifiers at level $n+1$. In turn, the classifiers at level $n+1$ can be regarded as instances of the classifiers defined at level $n+2$, and so on. Several of the OMG's metamodel specifications rely on 4 levels [OMG 06b]:

- M3. **Meta-metamodeling layer**, where the language for specifying metamodels is defined. MOF is an example of a meta-metamodel.

- M2. **Metamodeling layer**, where metamodels, the languages for specifying models, are defined. Examples of metamodels include the UML metamodel, the OCL metamodel, CWM, and the CCM. Metamodels are instances of meta-metamodels.

- M1. **Modeling layer**, where models are defined. User defined models in UML are among the most typical examples of a model. In general, models are used to define languages that specify the semantics of a domain. The models defined at M1 are instances of metamodels defined at M2. Note that these models also include illustrations, also referred to as snapshots, of instances of model elements (e.g. in UML, the object diagrams represent such illustrations).

- M0. **Run-time instances layer**, where instances of model elements are represented. These should not be confused with the illustrations of instances defined in level M1. The latter are constrained versions of the M0 run-time instances.

Figure 4.3 illustrates the distinctions among the 4 levels, using an example in UML. At the run-time instances layer (M0), we have a run-time instance of a thermometer. This thermometer is modeled at the user model level (M1) both as a class (`Thermometer`) and as a snapshot of the class (`:Thermometer`). In turn, the `Thermometer` class is an instance of the meta-class `Class`, of the UML metamodel (M2). The `temperature` attribute of the `Thermometer` class is an instance of the meta-class `Attribute`. The instance snapshot is modeled with the `Instance` meta-class.

Although the typical number of meta-levels ranges from 2 to 4, MOF-based metamodels can have at least 2 meta-levels, and as many meta-levels as one chooses to define. As we have seen, UML is an example of the usage of 4 meta-levels. The MOF specification [OMG 04] provides other examples of configurations, such as 2 meta-levels, for a generic reflective system, with classes and objects, and 3 meta-levels, for relational database systems, with `SysTables`, tables and rows.

The Object Constraint Language

MOF-based metamodels may include not only meta-elements, but also constraints applied to those meta-elements and to the meta-associations among them. These constraints are referred to as well-formedness rules, and can be formally specified using the Object Constraint Language (OCL) [OMG 03b]. OCL is a formal language designed to specify expressions on UML models that are typically used either as constraints on the model, queries on its state, or specifications of operations or actions, always in a programming language independent way. OCL expressions are side-effect free, so

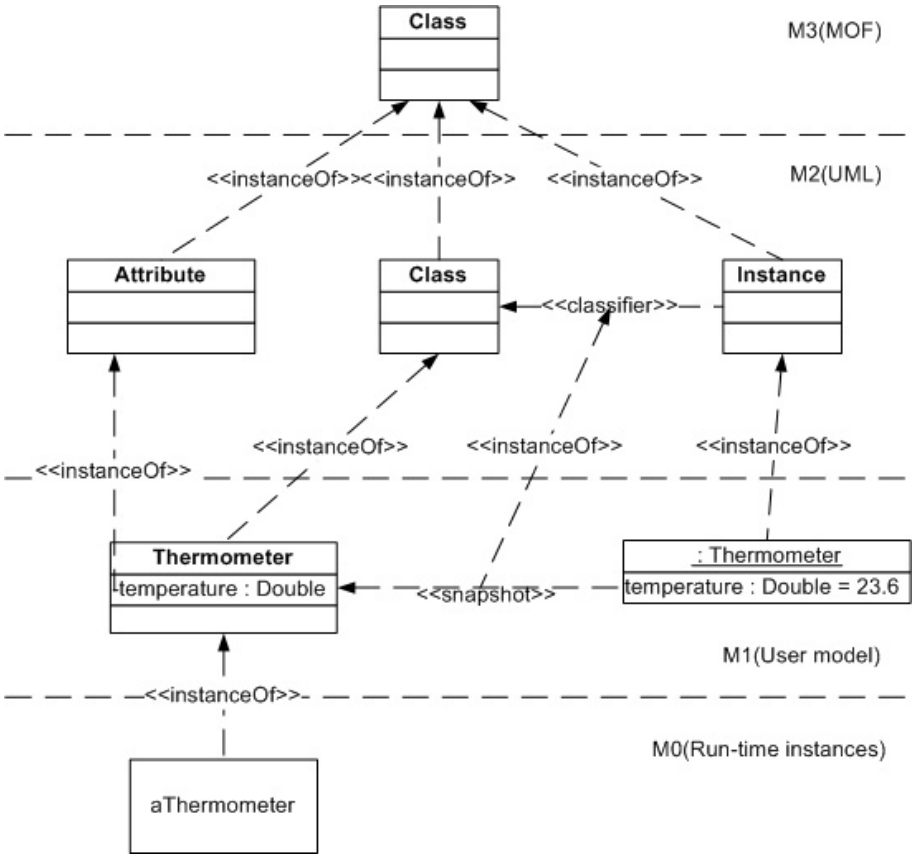


Figure 4.3: An example of the layered metamodel hierarchy

their evaluation does not alter the state of the system upon which they are defined, even if they specify operations that, when executed, would change the system state.

As MOF-based metamodels are defined using a subset of the UML language, OCL can be used on these metamodels. So, OCL is not only programming language independent, but also "MOF-based metamodel" independent.

To illustrate the usage of OCL as a constraint language, consider an extract of the UML 2.0 metamodel concerning the provided and required interfaces of a component. Figure 4.4 presents the associations, `provided` and `required`, between the meta-class `BasicComponent` and the meta-class `Interface`.

The `provided` and `required` associations are derived associations. The well-formedness rules of both derived associations are expressed in the UML metamodel both in natural language and with OCL. For illustration purposes, consider the specification of the derived provided interfaces association. Provided interfaces are *"the interfaces that the component exposes to its environment. These interfaces may be Realized by the Component or any of its realizingClassifiers, or they may be the Interfaces that are provided by its public Ports. The provided interfaces association is a derived association"* [OMG 05b]. In OCL, these constraints are expressed as in listing 4.1[2].

[2]This listing is extracted, from the UML 2.0 metamodel specification [OMG 05b]. Understanding all its details would require analyzing a larger extract of the UML 2.0 metamodel than the one presented in figure 4.4. Here, our purpose is only to illustrate some of the features of OCL

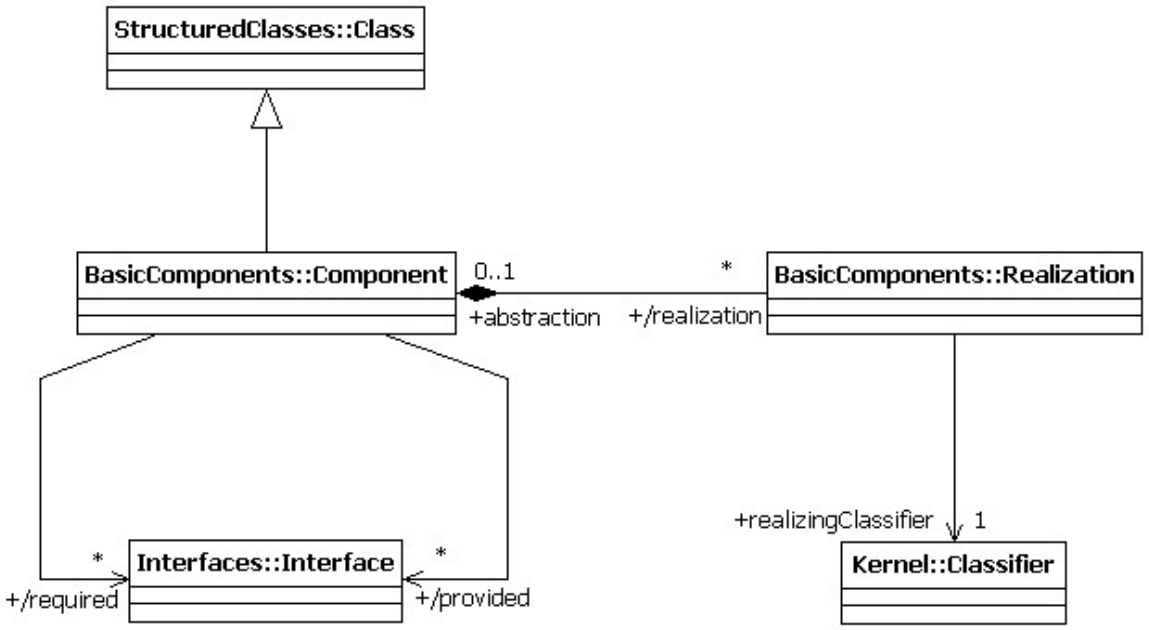


Figure 4.4: Extract of the UML 2.0 metamodel

Listing 4.1: An example of UML 2.0 well-formedness rules in OCL.

```
context Component::provided derive:
  let implementedInterfaces = self.implementation
      ->collect(impl|impl.contract) and
  let realizedInterfaces = RealizedInterfaces(self) and
  let realizingClassifierInterfaces =
      RealizedInterfaces(self.realizingClassifier) and
  let typesOfRequiredPorts = self.ownedPort.provided in

  (((implementedInterfaces->union(realizedInterfaces)
                          ->union(realizingClassifierInterfaces))
                          ->union(typesOfRequiredPorts))->asSet()

  def: RealizedInterfaces(classifier : Classifier) : Interface =
     (classifier.clientDependency->
          select(dependency|dependency.oclIsKindOf(Realization) and
          dependency.supplier.oclIsKindOf(Interface)))
               ->collect(dependency|dependency.client)
```

This well-formedness rule illustrates some features of the OCL which are relevant for our work. OCL expressions provide a side-effect free mechanism of extracting information from a model. OCL allows collecting information about the model (in this case, the UML 2.0 metamodel), while navigating through it. In this perspective, OCL can be viewed and used as a model query language. This property is useful for the collection of model elements according to any constraints we may specify with OCL. For instance, in the `Component::provided` well-formedness rule:

- `implementedInterfaces` is defined by selecting the collection of implementation associations and, for each of those, collecting all the corresponding contracts.
- `realizedInterfaces` is defined through an extra function, `RealizedInterfaces()`, also presented in listing 4.1.
- `realizingClassifierInterfaces` is also defined through the function `RealizedInterfaces()`.
- `typesOfRequiredPorts` is the union of the interfaces `provided` by the ports owned (`ownedPort` by the component.

Depending on the collection process, the resulting collection can be of different types, such as **bags**, or **sets**. OCL includes several utility operators on collections that allow, among other things, to make transformations between different kinds of collections. For instance, given a bag of integers, where integer values may be repeated, one can obtain the corresponding set of integers, where no duplicate values exist, using the function `asSet()`. We can see an example of this transformation in the derived rule `provide`. We can also flatten (i.e. recursively add elements of nested collections to a single collection) a collection, with the operation `flatten()`, or compute the number of elements in the collection, with the operation `size()`.

Besides being usable as a query language, the syntax of OCL is similar to that of programming languages, making it accessible for practitioners. Furthermore, OCL was designed to complement UML with a constraint specification language. Practitioners familiar with UML may also be familiar at least with some basic usage of the OCL language, e.g., through the definition of guard conditions in activity diagrams. Although historically the support provided by UML tools has been insufficient, UML 2.0 tools (e.g. Together Architect [3]) are progressively including support for the usage of OCL in their models.

In summary, OCL can be used as a query language for UML models, has a syntax that is accessible for practitioners and is increasingly being supported by UML tools. In our opinion, this combination of features makes OCL a well-suited candidate language for dealing with the metrics ill-definition problem. Practitioners can use a UML tool to specify the metamodel of the concepts which are relevant for their metrics collection, load that metamodel with the model instances representing the artifacts they aim to measure, and use OCL not only to specify, but also to collect metrics on those models. The necessary context for defining metrics is provided by the specified metamodel. The metrics definitions are not ambiguous, as they are specified through OCL constraints. We can also specify the conditions under which the metrics can be computed, through pre-conditions expressed in OCL. Finally, the usage of standardized languages for defining the measurement context (a metamodel defined with UML)

[3] `http://www.borland.com/us/products/together/index.html`

and the metrics (through OCL functions) facilitates the replication of metric collection initiatives, thus fostering the independent validation of metrics.

4.3 Defining and collecting metrics with OCL

In this section we will define two simple metrics for counting the number of attributes and operations of a software component. We will use the UML 2.0 metamodel as our ontology. In other words, the metrics formalization described in this section will allow counting the number of attributes and operations of a UML 2.0 component.

Before defining the metrics, we have to study how UML 2.0 represents components. Consider the extract of the UML 2.0 metamodel presented in figure 4.5. As discussed before, this diagram represents part of a specification at the M2 level. For the sake of readability, to avoid cluttering this class diagram in excess, most of the attributes of the meta-classes presented in this diagram are omitted. In this metamodel extract we can observe that a `Component` is a specialization of `Class`, as defined in the `StructuredClasses` package. The UML 2.0 metamodel includes several meta-classes called `Class` (this extract only shows 5 of them), each defined in a different package, which incrementally add specific details to the abstract notion of `Classifier`. In the UML 2.0 metamodel, `Classifier` is used as a basis to one of the three major categories of elements (the remaining two categories are events and behaviors). Each `Component` has a set of properties (`ownedProperty`) and provides a set of operations (`ownedOperation`). Operations may contain parameters (`ownedParameter`).

Consider the `SQL_Select` component, in figure 4.6, defined in a user model (M1 level). This fine grained component contains 9 attributes and 21 operations. Out of those 21 operations, 1 is a constructor, 5 are getter operations, 5 are setter operations, and the remaining 10 are business operations (all the operations with no stereotype, in the example) [4].

This component can be represented by instantiating the UML 2.0 metamodel with appropriate meta-objects and meta-links. Figure 4.7 presents a small extract of the meta-objects diagram that results from that instantiation. Again, for the sake of readability, several meta-objects and meta-links are omitted. This snapshot is at the M1 level.

4.3.1 Using OCL expressions to collect information

The following OCL expressions compute the set of owned properties, its size, the set of owned operations, and its size, respectively. The two size expressions, in particular,

[4] We use the term "business operations" to conform to Washizaki *et al.*'s classification [Washizaki 03] for all the available operations that implement the functionality of components other than constructors, "getters" and "setters". For the time being this distinction is not relevant, but we will refer to it in the case study in section 4.4.

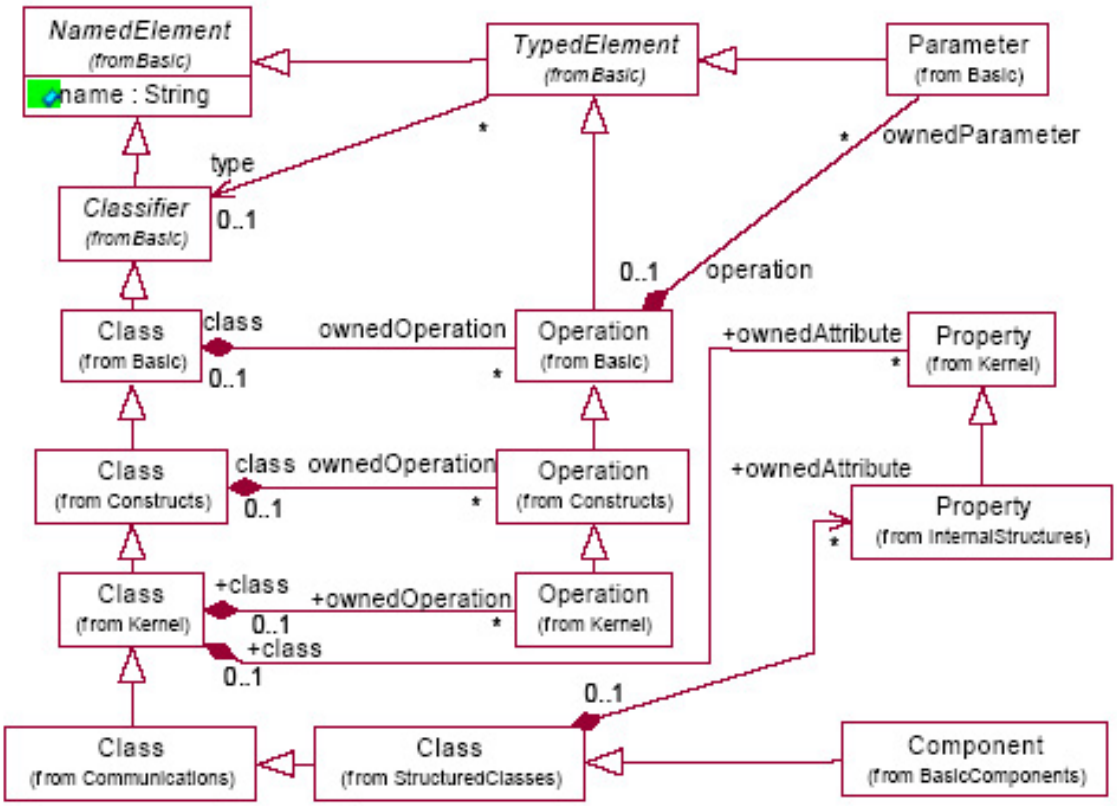


Figure 4.5: UML 2.0 metamodel extract

show how simple measurements can be computed for a given component. The results of the evaluation of each of the expressions are also presented. In the OCL listing 4.2, the lines started by '?' represent OCL queries. The corresponding answers are presented in *italic*. For the sake of simplicity, we assume the objects that populate the metamodel have the same name as the model elements they represent, in these OCL queries.

Listing 4.2: Making queries in OCL.

```
?SQL_Select.ownedAttribute
{NO_WORK, ..., maxRows}
?SQL_Select.ownedAttribute->size()
9
?SQL_Select.ownedOperation
{SQL_Select, ..., writeObject}
?SQL_Select.ownedOperation->size()
21
```

In OCL it is possible to define clauses within a given context. The set of clauses in listing 4.3 is defined for elements of the meta-class `Component`. They allow computing the number of owned attributes `A()` and the number of owned operations `O()`. The `self` keyword denotes the object receiving a method call. The `size()` operations counts the elements in the collection.

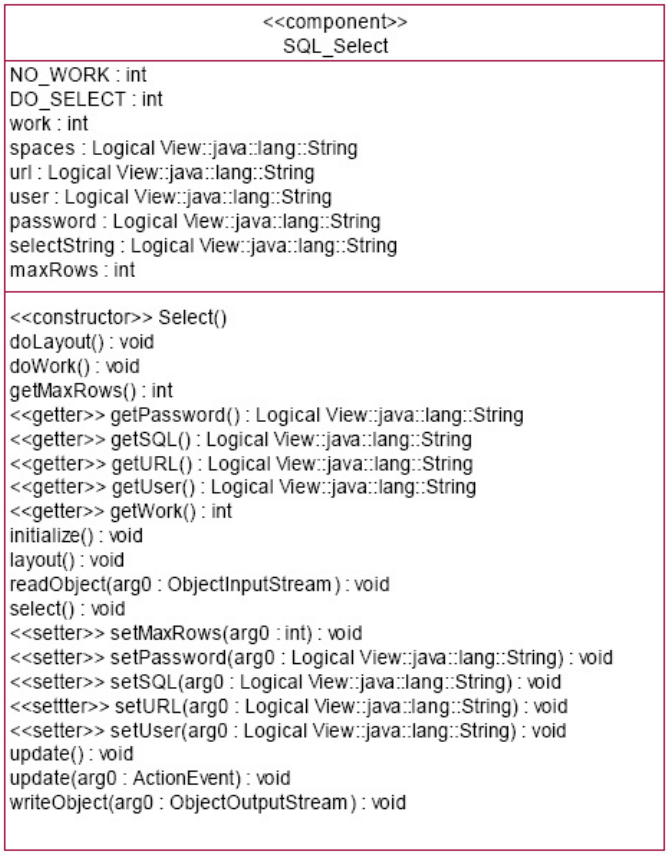


Figure 4.6: The SQLSelect component

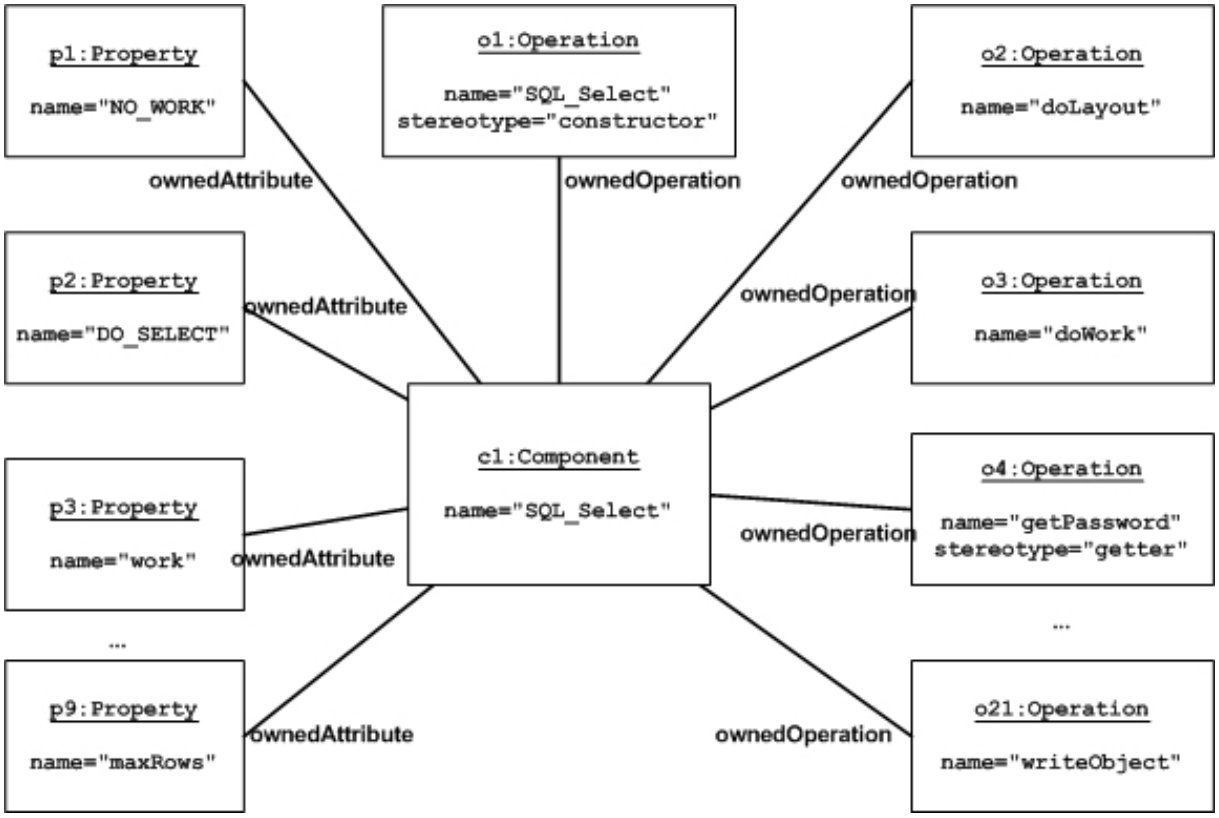


Figure 4.7: The SQLSelect component instantiation

Listing 4.3: Defining how to count owned attributes and operations.

```
context Component
A(): Integer = self.ownedAttribute->size()
O(): Integer = self.ownedOperation->size()
```

Using these OCL expressions, we can now obtain, for the `SQL_Select` component, the number of properties, respectively (listing 4.4).

Listing 4.4: Counting owned attributes and operations.

```
?SQL_Select.A()
9
?SQL_Select.O()
21
```

4.4 The FukaBeans case study

4.4.1 Motivation

To illustrate the ODM approach, we present here a formalization of a metrics set proposed by Washizaki *et al.* in [Washizaki 03]. Our formalization is used in a case study where we collect metrics from *JavaBeans* components and use those metrics to assess the components against a component quality model. As a quality model, we will use the component quality model defined by Washizaki *et al.*, when proposing and validating their metrics set [Washizaki 03]. The components analyzed in our case study were not included in the components set used in the validation experiment conducted by Washizaki *et al.*

Note that the main purpose of this case study is to illustrate how ODM can be used in practice, rather than thoroughly validating Washizaki *et al.*'s quality model. So, our emphasis will be more focused on the metrics definition technique than on the independent validation itself. Nevertheless, we will follow the experimental process described in chapter 3.

Problem statement

Washizaki *et al.* proposed a quality model for reusability of JavaBeans components, and a metrics set for helping to assess whether or not a given JavaBean is reusable. In this case study we will perform an independent validation for their model, and the metrics. We can briefly state our main hypothesis as follows:

H1 Washizaki *et al.*'s quality model and metrics are valid, as suggested in the paper where they were proposed.

Research objectives

In this case study our goal is to:

analyze JavaBean components,
for the purpose of their evaluation
with respect to their reusability, as defined in [Washizaki 03]
from the viewpoint of the quality auditor,
in the context of a JavaBeans component library called *FukaBeans* [Fukazawa 03].

Context

As this is an independent validation effort, our main objective is to check the extent to which the quality model holds, when using a different set of components. While the original validation effort used a sample of COTS components, we will use a sample of components developed for educational purposes. In both cases, the assumption is that the components in both sets were designed to be highly reusable. The rationale for this assumption is different: in the original experiment, expert opinion was used for assessing the reusability of components. In our replica, the sample is made of components developed for educational purposes, in a context where high reusability is one of the main quality concerns. So, we can think of this as a differentiated replica of the original validation effort.

This case study is conducted off-line, using artifacts which can be considered to be toy components, when compared to those used in the original validation effort. The generalizability of the results in the case study is restricted to the set of components under scrutiny.

4.4.2 Related work

The quality model

The quality model used by Washizaki *et al.* is presented in figure 4.8. This quality model breaks component **reusability** down into three quality attributes: **understandability**, **adaptability**, and **portability**. The three quality attributes are broken into four assessment criteria, which are then mapped into five metrics for *JavaBeans* component reusability assessment.

The metrics set

The metrics set includes 5 reusability metrics, as depicted in figure 4.8: `EMI`, `RCO`, `RCC`, `SCCr`, and `SCCp`. We will discuss each of these metrics definition in detail, while presenting their formalization. For each of the metrics, Washizaki *et al.* describe:

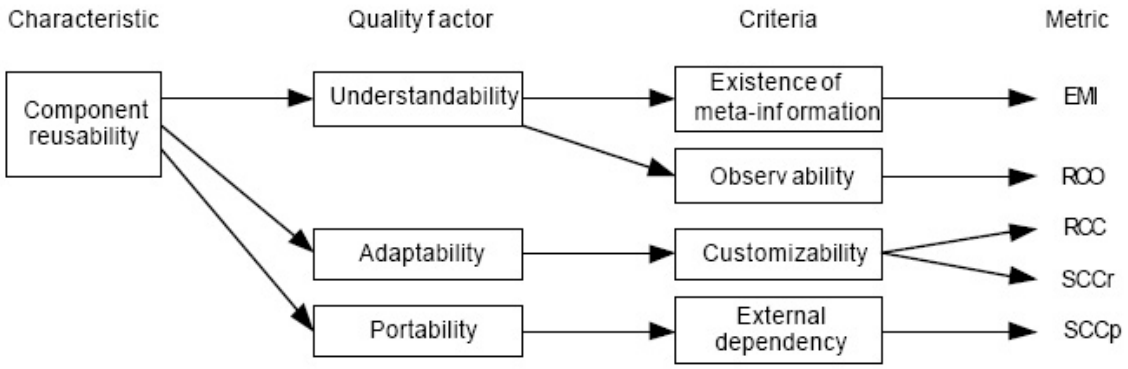


Figure 4.8: The quality model used by Washizaki *et al.*

- their intent;
- their definitions, combining mathematical formulas with informal descriptions of the elements used in those formulas;
- a confidence interval [Lower Limit; Upper Limit] for each metric; it is assumed that components that have metrics values outside of this confidence interval are flagged as a potentially less reusable than those components which are not flagged;
- a set of heuristics to facilitate the interpretation of the measurement values (thus making this a metrics set), based on the earlier mentioned confidence intervals.

We will frame each of the metrics description in the context of the GQM approach, while presenting their formalization.

Validation efforts for this quality model

To the best of our knowledge, the work described here (previously published in [Goulão 04a, Goulão 05c]) is the only independent validation for this quality model and metrics. Both validations are framed by the same quality model (the one used in Washizaki *et al.*'s paper).

Our validation effort differs from the one followed by Washizaki *et al.* in the following aspects:

- The sample of components used in both validation efforts differs in:
  - the size of the sample;
  - the typical complexity of components;
  - the origin of components;
  - the rationale for the assumption of high reusability, as observed externally without using the quality model.

- The metrics definition technique used in Washizaki *et al.*'s work was a combination of an informal description of JavaBeans with mathematical formulas. We formalize their metrics in OCL using the UML 2.0 as a metamodel for representing the JavaBeans components. As we will discuss, this has some implications with respect to the accuracy of our metrics definitions, as some decisions were made to solve ambiguities in the original metrics definition.

4.4.3 Experimental planning

We can break down our original goal into three sub-goals, where the variation lies on the *quality factor* under scrutiny in each sub-goal. For each of these sub-goals, we will also present the research question we would ask to assess them, as well as the used quality criteria and metrics.

Goal 1(G1):
Analyze JavaBean components
(...)
with respect to their ***understandability***
(...)

Goal 2(G2):
Analyze JavaBean components
(...)
with respect to their ***adaptability***
(...)

Goal 3(G3):
Analyze JavaBean components
(...)
with respect to their ***portability***
(...)

Experimental units, material and tasks

In this case study we are just collecting data from already existing components, rather than asking subjects to perform a given set of tasks, upon some sort of material. So, we have no experimental units, material, or tasks.

Hypotheses

We can break down the goals of this case study into questions, and, for each of the later, define a corresponding hypothesis. The notation, for easier reference, is that Goal 1 (G1) corresponds to Question(s) 1.*. Question Q1.1 corresponds to hypothesis H1.1, which will then be assessed with the support of metric M1.1.

Question 1.1 (Q1.1):
Is meta-information available for the component?

Hypothesis 1.1 (H1.1):
Components adhering to the *understandability* standards of the quality model provide **meta-information** for their users.

Question 1.2 (Q1.2):
Is the level of component observability adequate?

Hypothesis 1.2 (H1.2):
Components adhering to the *understandability* standards of the quality model have an **observability** within the boundaries established by it.

Question 2.1 (Q2.1):
Is the level of component customizability adequate?

Hypothesis 2.1 (H2.1):
Components adhering to the *adaptability* standards of the quality model have a **customizability** within the boundaries established in the quality model.

Question 3.1 (Q3.1):
Is the level of *external dependency* of a component with respect to other components acceptable?

Hypothesis 3.1 (H3.1):
Components adhering to the *portability* standards of the quality model have a relative number of external dependencies within the boundaries established in the quality model.

Question 3.2 (Q3.2):
Is the number of dependencies to external components which stem from the return types of the component's operations acceptable?

Hypothesis 3.2 (H3.2):
Components adhering to the *portability* standards of the quality model have a relative number of business operations with parameters within the quality model's thresholds.

Independent variables

The independent variables used in this case study will be Washizaki *et al.*'s metrics. There is a direct correspondence between each hypothesis and its corresponding metric (e.g. H1.1 corresponds to Metric 1.1 (M1.1)). For each of these metrics, we will start by an informal description, followed by the corresponding mathematical formula, before providing the metric formalization.

Metric 1.1 (M1.1): Existence of Meta-Information (`EMI`).
The *JavaBeans* component model relies on the existence of a special `BeanInfo` class to provide access to meta-information about the *JavaBean* component. The correspondence between a *JavaBean* and its `BeanInfo` is established by using a naming convention that appends the `BeanInfo` string to the name of the class. For example, a component named `Car` would have a `CarBeanInfo` class for representing its meta-information. `EMI` of a component `c` is defined as:

$$EMI(c) = \begin{cases} 1 & \text{, if the BeanInfo class exists} \\ 0 & \text{, otherwise} \end{cases}$$

Metric formalization: The notion of a *BeanInfo* class is not conveniently represented in the UML metamodel, as it is specific to the *JavaBeans* component model. The presence of such a *BeanInfo* is detected through a naming convention by *JavaBeans*. The usage of a naming convention, not captured by the UML 2.0 metamodel, creates a dilemma for the formalization of this metric's definition: neither the UML 2.0 metamodel includes structural information on the implicit relationship between a class and its meta-information, nor OCL has primitives for handling the contents of a string (therefore, encoding the naming convention detection in standard OCL is not practical).

Two alternatives for enabling the formalization of this metric would be to:

- Extend the UML 2.0 metamodel, to support the concept of the *BeanInfo* class, and its relations to other classes in the metamodel.
- Extend the OCL with primitives for handling strings. Such an extension is available, for instance, in the Together Architect tool, with a set of convenient String handling functions that can be used seamlessly in the OCL expressions.

For illustration purposes, we include the definition of the EMI metric, using Together Architect's extended OCL, although the reader should note that this definition

does not adhere to the standard OCL. The OCL extension in Together Architect allows using string handling functions such as `append(...)` to collect information about a string (listing 4.5).

Listing 4.5: EMI formalization in OCL, using Together Architect's OCL extension.

```
context Component
-- Existence of Meta-Information
EMI(): Real =
     if (Component->allInstances->
                  includes(self.name.append('BeanInfo'))) then
         1.0
     else
         0.0
     endif
```

Metric 1.2 (M1.2): Rate of Component Observability (`RCO`).
The observability of a component depends on the percentage of component properties that are accessible for reading from outside of the component. There is a tension between the need to make relevant information available, while keeping a good encapsulation level for the component. Providing unnecessary access to properties increases the complexity of the component interface, decreasing its understandability. Conversely, providing insufficient access to component properties may also decrease its understandability. `RCO` of a component `c` is defined as:

$$RCO(c) = \begin{cases} \frac{P_r(c)}{A(c)} & \text{, if A(c)>0} \\ 0 & \text{, otherwise} \end{cases}$$

where:
$P_r(c)$: number of properties that are accessible for reading in `c`.
$A(c)$: number of properties in `c`.

To facilitate the formalization of `RCO`, we start by defining two auxiliary functions. `A()` is the number of accessible properties in a component. `Pr()` is the number of properties accessible for reading in the component. Note that Washizaki *et al.* use the number of owned operations stereotyped with `<<getter>>` as a surrogate for the number of properties. Finally, we define `RCO()` as a ratio between `Pr()` and `A()`.

Listing 4.6: RCO formalization in OCL.

```
context Component
-- Number of Readable Properties
Pr(): Integer =
    self.ownedOperation->select(o: Operation|
        o.stereotype = 'getter')->size()

-- Number of Properties in the component
A(): Integer  =
```

```
   self.ownedAttribute->size()

-- Rate of Component Observability
RCO(): Real = if self.A() = 0 then
                 0.0
              else
                 self.Pr()/self.A()
              endif
```

Metric 2.1 (M2.1): Rate of Component Customizability (RCC).
The percentage of the properties of a component that are available for writing is related to the customizability of a component. As it happens with the access to readable properties, it is desirable to achieve a balance between the flexibility offered by the component in its customization and the effort required from the component user to parameterize the component. RCC can be defined as follows:

$$RCC(c) = \begin{cases} \frac{P_w(c)}{A(c)} & \text{, if A(c)>0} \\ 1 & \text{, otherwise} \end{cases}$$

where:
$P_w(c)$: number of properties that are accessible for writing in c.
$A(c)$: number of properties in c.

The formalization of this metric uses an auxiliary function, Pw(), that computes the number of writable properties in the component. These are defined by Washizaki *et al.* as the number of setter operations. Of the operations owned by a component, we select those which are stereotyped with <<setter>> (listing 4.7).

Listing 4.7: RCC formalization in OCL.

```
context Component
-- Number of Writable Properties
Pw(): Integer =
    self.ownedOperation->select(o: Operation|
       o.stereotype = 'setter')->size()

-- Rate of Component Customizability
RCC(): Real = if self.A() = 0 then
                 0.0
              else
                 self.Pw()/self.A()
              endif
```

Metric 3.1 (M3.1): Self-Completeness of Component's parameters(SCCp).
Portability of components is related to the level of their external dependencies that result from the parameters available in their business operations. The percentage of

parameters that are used in the component's business operations that are external to the component specification provides an indirect measure of the dependency of that component to other components. Components with too many dependencies may be less portable than components with fewer dependencies, because they require the indirect reuse of more components.

$$SCC_p(c) = \begin{cases} \frac{B_p(c)}{B(c)} & \text{, if B(c)>0} \\ 1 & \text{, otherwise} \end{cases}$$

where:
$B_p(c)$: number of business operations with parameters in c.
$B(c)$: number of business operations in c.

In OCL, this can be formalized as in listing 4.8:

Listing 4.8: RCC formalization in OCL.

```
context Component
-- Set of business methods provided by the component
BusinessMethods(): Set (Operation)=
    self.ownedOperation->select(o: Operation|
       (not (o.stereotype = 'constructor')) and
       (not (o.stereotype = 'getter')) and
       (not (o.stereotype = 'setter')))

-- Number of business methods
B() : Integer =
    self.BusinessMethods->size()
```

Metric 3.2 (M3.2): Self-Completeness of Component's return values(SCCr).
Portability of components is related to the level of their external dependencies that result from the return values of the business operations of the components. The percentage of return value types that are used in the component's business operations that are external to the component specification provides an indirect measure of the dependency of that component to other components. Again, components with too many dependencies may be less portable than components with fewer dependencies, because they require the indirect reuse of more components.

$$SCC_r(c) = \begin{cases} \frac{B_v(c)}{B(c)} & \text{, if B(c)>0} \\ 1 & \text{, otherwise} \end{cases}$$

where:
$B_v(c)$: number of business operations with parameters in c.
$B(c)$: number of business operations in c.

The formalization in OCL is performed in two different classes: Component and

`Operation` (see listing 4.9).

Listing 4.9: Business methods counting formalization in OCL.

```
context Component
-- Number of business methods with no return value
Bv(): Integer =
    self.BusinessMethods()->select(b: Operation|
       b.ReturnTypeName()= 'void')->size()

-- Number of business methods with no parameters
Bp(): Integer =
    self.BusinessMethods()->select(b: Operation|
       b.OwnedParameter()->size() = 0)->size()
```

The set of operations in listing 4.10 is defined for elements of the type `Operation`.

Listing 4.10: Auxiliary functions formalization in OCL.

```
context Operation
-- Set of formal parameters(except return parameter)
Params(): Set(Parameter) =
    self.formalParameter->select(fp: Parameter |
      fp.direction <> #return)

-- Set of return parameters of an Operation
ReturnParams(): Set(Parameter) =
    self.formalParameter->select(fp: Parameter |
      fp.direction = #return)

-- Return type name of an Operation
ReturnTypeName (): String =
    if (self.formalParameter ->
        exists(direction = #return))
    then if (self.ReturnParams()->asSequence()->
        first.type.isDefined)
         then
            self.ReturnParams()->asSequence()->
               first.type.name
         else
            'void'
         endif
    else
         'void'
    endif
```

Dependent variable

The dependent variable in this case study is reusability. It is assumed that all the elements of our sample exhibit the property of being highly reusable.

So, as the assumption is that all elements in the sample are highly reusable, we will now formalize the definition of heuristics in OCL, which is used to assess whether or not a component is highly reusable.

Washizaki *et al.*'s proposal includes a set of component design heuristics based on these metrics. Three of the heuristics, from here on referred to as `WarningRCO`, `WarningRCC`, and `WarningSCCp`, based on the `RCO`, `RCC`, and `SCCp` metrics, respectively, can be regarded as band-pass filters. They establish low and high thresholds for the corresponding metrics values. If the metrics values go beyond those thresholds, a design warning will be triggered.

The `WarningSCCr` and `WarningEMI` heuristics are high-pass filters: they define a low threshold for the metric (`SCCr`, or `EMI`, respectively). If the metric value is below this threshold, a design warning will be triggered, in this case warning of a potential lack of adaptability (`WarningSCCr`) or of meta-information (`WarningEMI`).

To formalize these heuristics, we start by defining the three different heuristics kinds: low-pass filter (`AboveRange`), high-pass filter (`BelowRange`), and band-pass filter (`OutOfRange`). We define these functions at the `class` meta-class, so that we can reuse them with any model element. In this example, the heuristics that will reuse these functions are defined upon the `Component` meta-class (listing 4.11).

Listing 4.11: Heuristics rules formalization in OCL.

```
context Class
-- High-pass filter
AboveRange (limit: Real, value: Real): Boolean =
   value > limit

-- Low-pass filter
BelowRange (limit: Real, value: Real): Boolean =
   value < limit

-- Band-pass filter
OutOfRange (lowerLimit: Real, upperLimit: Real,
            value: Real): Boolean =
   (self.BelowRange (lowerLimit, value))
   or (self.AboveRange (upperLimit, value))
pre: lowerLimit < upperLimit

context Component
-- RCO-based heuristic.
WarningRCO(lowerThreshold: Real,
           upperThreshold: Real): Boolean =
   self.OutOfRange (lowerThreshold, upperThreshold,
   self.RCO())
pre: lowerThreshold < upperThreshold

-- RCC-based heuristic.
WarningRCC(lowerThreshold: Real,
```

```
            upperThreshold: Real): Boolean =
   self.OutOfRange (lowerThreshold, upperThreshold, self.RCC())
pre: lowerThreshold < upperThreshold

-- SCCr-based heuristic.
WarningSCCr(lowerThreshold: Real): Boolean =
   self.BelowRange(threshold, self.SCCr())

-- SCCp-based heuristic.
WarningSCCp(lowerThreshold: Real,
            upperThreshold: Real): Boolean =
   self.OutOfRange (lowerThreshold, upperThreshold, self.SCCp())
pre: lowerThreshold < upperThreshold
```

We can now define a `DesignWarning` heuristic that combines all the others, using the disjunction operation.

Listing 4.12: Heuristics rules formalization in OCL.

```
context Component
DesignWarning(
   RCO_LL:  Real, RCO_UL:  Real,
   RCC_LL:  Real, RCC_UL:  Real,
   SCCp_LL: Real,
   SCCr_LL: Real, SCCp_UL: Real,
   EMI_LL: Real, EMI_UL: Real): Boolean =
         (self.WarningRCO(RCO_LL, RCO_UL))
      or (self.WarningRCC(RCC_LL, RCC_UL))
      or (self.WarningSCCr(SCCr_LL))
      or (self.WarningSCCp(SCCp_LL, SCCp_UL))
      or (self.WarningEMI(EMI_LL, EMI_UL))
```

In order to use this heuristic, we have to establish the heuristic's parameters. These correspond to the low and high limits of each of the individual heuristics. Table 4.1 presents the thresholds for each of the metrics, taken from the results of the validation experiment conducted by Washizaki *et al.* [Washizaki 03]. From left to right, we can observe the name of the metric, the mean value found in the sample, the lower threshold for each metric, the upper threshold, and the number of components within the sample that were inside the interval between the lower and upper thresholds. Note that two of the metrics have 1.00 as their upper threshold. As the metrics on this set have values in the interval $[0.00, 1.00]$, an upper threshold of 1.00 means, in practice, that the heuristic for the corresponding metric has only a lower threshold (so, we can think of this as a high-pass filter). The remaining three heuristics use both a low and a high threshold for their respective metrics (we can think of these metrics as band-pass filters).

Washizaki *et al.* computed these limits using a sample of 125 *JavaBeans* available

Metric	Mean	Lower limit	Upper limit	#components
RCO	0.40	0.17	0.42	36
RCC	0.35	0.17	0.34	35
SCCr	0.85	0.61	1.00	108
SCCp	0.74	0.42	0.77	28
EMI	0.84	0.50	1.00	105

Table 4.1: Metrics heuristics thresholds.

from a commercial broker [5]. In the component broker web site, components were assessed by a panel of experts that rated the components according to evaluation criteria such as **presentation**, **functionality** and **originality**. The resulting ranking in the broker web site was a classification between 0 and 1000 points. Washizaki *et al.* stipulated that components with more than 875 points would be classified as "highly reusable" and then used those components to establish the thresholds presented in table 4.1, with a degree of confidence of 95%. Concerning the `WarningEMI` thresholds, note that EMI can only assume the values 1 or 0. Therefore, the lower limit value could have been set at any arbitrary value in the interval $0 < threshold < 1$. The chosen value was the middle of that interval (0.50).

When using statistics-based heuristics to assess components, we must stress that the heuristics only aim to help detecting, with a high probability, the presence of a problem. We can not ignore that there is a low probability of the problem existing, even if the metrics values are within the allowed limits of the confidence intervals. Conversely, there is also a low probability of a high quality component presenting metrics values that suggest it has a poor quality. This concern is typical from statistics-based approaches to assessment, and is usually referred to **type I errors** (also known as false positives) and **type II errors** (false negatives), respectively.

Design

In this case study, we will use a single group of subjects (the components) and a single observation. The assumption is that all the subjects of the sample share a property (high reusability)[6].

Procedure

As discussed earlier, it is important to have an automated collection of metrics. The tool requirements for the application of ODM with UML 2.0 are as follows:

- the tool should provide full UML 2.0 modeling capabilities;

[5] `http://www.jars.com/`

[6] Note that, for a more thorough cross-validation effort, a more sophisticated experiment design should be used. However, in this chapter, we are mostly concerned with putting ODM in the context of the experimental process described in chapter 3, so we decided to keep the case study very simple.

- the tool should use the complete UML 2.0 metamodel as its underlying model. In other words, the tool's repository should be UML 2.0 compliant. This implies that models (M1) would be stored as metamodel (M2) instances;
- The tool should provide the ability of computing the values of OCL expressions at the metamodel level (M2), as well as at lower levels.

To the best of our knowledge, no current tool completely supports all these features. Some of the tools that support OCL expressions specification and computation allow writing those expressions upon a specific model, rather than at the metamodel level (e.g. OCLE[7]). Others allow using those expressions on its metamodel, as well (e.g. Together Architect), but use a non-standard metamodel. It is common for tools to use a simplified version of the UML metamodel, rather than its full specification, as the tool's repository model. The implication of the usage of these specifications is that even if OCL expressions can be written and evaluated at the metamodel level, the metamodel itself is non-standard, making the OCL expressions invalid for the specific metamodels used by other tools.

In this case study, we opted for the usage of a collection of tools that mixes off-the-shelf tools with custom made ones. The architecture is depicted in figure 4.9. The XMI front-end is a custom-made tool that transforms UML models in XMI in USE models (providing the meta-classes) and USE instantiation scripts (providing the meta-objects). The UML metamodel may be obtained from the OMG web site [8], while the XMI for the component assembly specification (which, in this case, contains the specification of the *JavaBeans* components) can be generated with a UML 2.0 tool with JavaBeans reverse engineering to UML, and XMI generation from a loaded model. The USE tool [9] [Richters 01] is loaded with the metamodel, the metrics definitions, the OCL heuristics specification, and the meta-objects representing the JavaBeans. OCL queries performed upon these meta-objects produce the metrics and heuristics results, which can be stored in text files, for further analysis by the researcher. In particular, these text files can be generated as Comma Separated Values (CSV) files, which are convenient if we want to import them and perform further analysis using a spreadsheet, or a more sophisticated statistics tool.

Analysis procedure

As the main purpose of this case study is to illustrate ODM, and how it can fit into the experimental process described in chapter 3, rather than the independent validation of the quality model and metrics, we used a fairly small sample of components and will only make a relatively simple analysis on some descriptive statistics collected from this sample. Although more powerful statistics could be used, the characteristics of

[7] http://lci.cs.ubbcluj.ro/ocle/index.htm
[8] http://omg.org/cgi-bin/doc?ptc/04-10-05.zip
[9] http://www.db.informatik.uni-bremen.de/projects/USE/

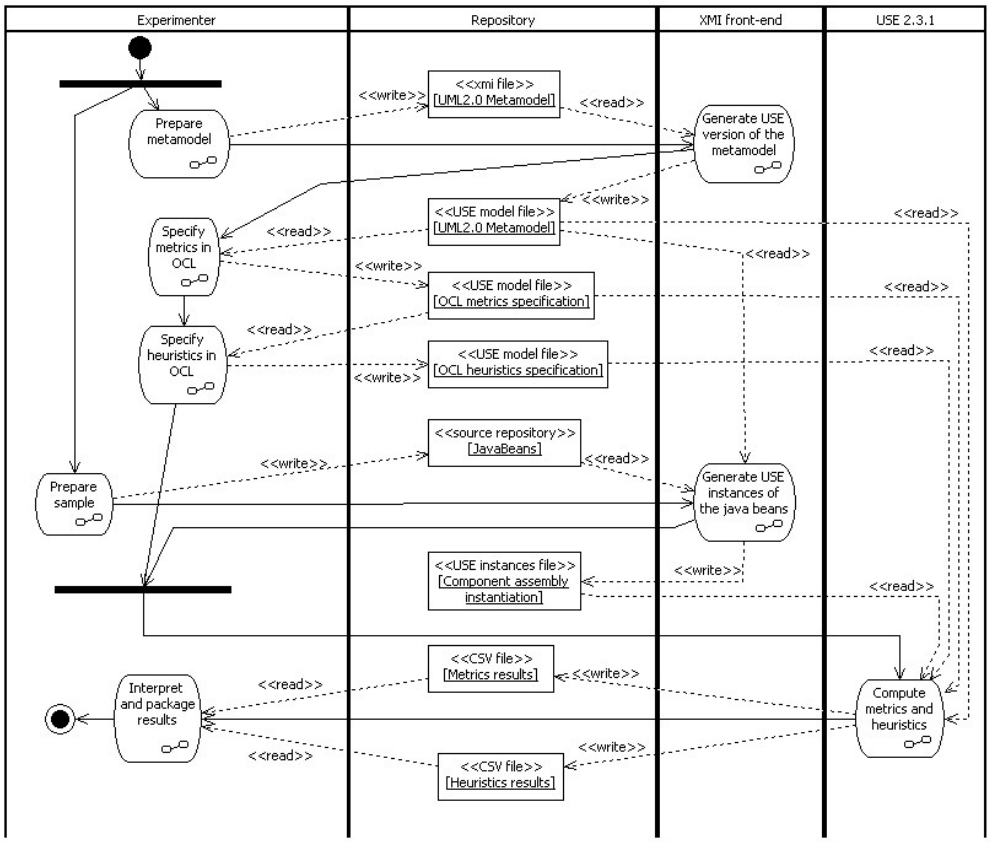


Figure 4.9: Data collection and analysis

the sample would create several threats to the validity of the results. We will make an analysis concerning how well the scrutinized components behave, with respect to the heuristics specified earlier in this chapter.

4.4.4 Execution

Sample

We used the metrics set with the FukaBeans component library [Fukazawa 03], a public domain *JavaBeans* library developed, with educational purposes, by members of Washizaki's research team. Each component is distributed as a separate *jar* archive. The sample includes 12 components.

Preparation

No special preparations were required, other than downloading the JavaBeans. The jar archives were analyzed as they were, without transformations.

Data collection performed

The data collection followed the plan outlined in the previous sub-section (more specifically, in the Procedure sub-sub-section).

4.4.5 Analysis

Descriptive statistics

Table 4.2 presents the metrics results, excluding the EMI metric[10] The values in **bold** represent violations to the previously defined thresholds. The table also includes the mean value and standard deviation for each of the metrics, in the bottom.

JavaBean	RCO	RCC	SCCr	SCCp
CellBean	**0,037**	**0,111**	0,909	**0,818**
FileUtil	**1,000**	**0,667**	1,000	**1,000**
FilterBean	0,267	**0,133**	0,933	**0,200**
FukaCalendarBean	**0,444**	**0,444**	0,857	0,571
FukaGraphBean	**1,000**	**1,000**	1,000	0,733
FukaStopWatchBean	**0,667**	**0,667**	1,000	**0,200**
FukaTextBean	**0,000**	**0,000**	1,000	**1,000**
GameBean	0,250	0,250	1,000	0,556
GraphBean	0,182	0,273	1,000	0,714
StatementBean	**0,667**	**0,667**	**0,500**	0,500
DocumentBean2	**0,000**	**0,000**	1,000	**1,000**
WordBean2	**0,000**	**0,000**	1,000	**1,000**
Mean	*0,376*	*0,351*	*0,933*	*0,691*
Std.Dev.	*0,376*	*0,333*	*0,145*	*0,294*

Table 4.2: Metrics collected on the *FukaBeans* component library.

Although the average values for the metrics are within the quality intervals established by Washizaki *et al.*, only two of the components (`GameBean` and `GraphBean`) comply with all the quality heuristics. 7 out of the 12 assessed components were signaled by 3 out of the 4 used heuristics.

In table 4.1 we presented the heuristics thresholds. Then, in table 4.2 we presented the metrics values, collected from the FukaBeans components.

An alternative form of presenting this data is to have a graphical representation of the heuristics, using a Kiviat diagram, such as the one presented in figure 4.10. To avoid cluttering the diagram, we only present two of the components (`CellBean` and `FukaCalendarBean`), along with the low and high thresholds.

This kind of visualization provides a faster (and simpler) way to convey the information to practitioners developing components, by highlighting potential areas for improving their components (through the values which are out of the recommended boundaries).

[10] `EMI` was not collected, because we decided to use just the standard OCL in this metrics collection, rather than any custom-made extension, such as the one available in Together Architect, and discussed earlier in this chapter, while introducing the `EMI` metric.

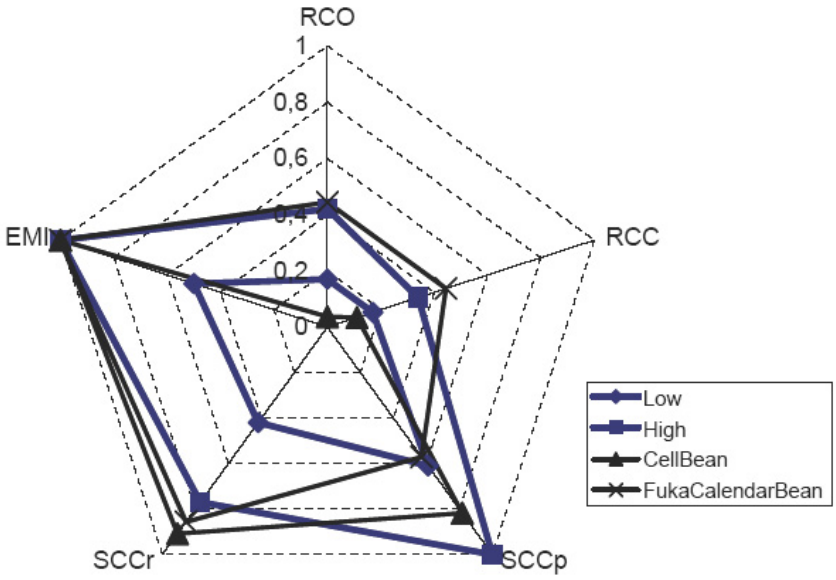


Figure 4.10: Quality model thresholds Kiviat diagram

Data set reduction

No components were removed from the sample.

Hypothesis testing

As discussed in the beginning of sub-section 4.4.5, we did not carry out a statistical test of hypotheses in this case study, as the sample is too small for the results to be meaningful.

4.4.6 Interpretation

Evaluation of results and implications

It may seem surprising that most of the components in a library developed by the metrics proponents fail to meet the thresholds included in the metrics set proposal. The thresholds used in this case study were derived from a sample of a population frame which consisted of COTS components. The sample used here includes components developed for educational purposes, with a typically low complexity (their implementation has less than 10 methods). The apparent lack of reusability of most of the components, according to the heuristics, clashes with the expectation that components developed with educational purposes would be well designed, to foster the utilization of component development best practices by students.

Threats to validity

Perhaps the major threat to the validity of this case study concerns the characteristics of the used sample, and their implications on the kind of statistical analysis that would

make sense to perform. With only 12 components to be analyzed, it is dangerous to extrapolate definitive conclusions from this case study.

Another issue concerns the single group, single observation design of this replication. All subjects are assumed to be of a high reusability, which makes it impossible to test whether or not components with a low reusability would also meet the quality thresholds of Washizaki *et al.*'s quality model.

Inferences

The identified threats to the validity of the case study do not imply that the effort of conducting exploratory case studies such as this one is fruitless. It is beyond the scope of this dissertation to conduct a comprehensive effort for the validation of this quality model and metrics set. With all the care that must be taken when inferencing based on an exploratory case study, the one described in this chapter does point to a potential problem in this metrics set and its underlying quality model, concerning the thresholds proposed by Washizaki *et al.*.

When experimentally validating statistical models, one should cover, not necessarily on a single experiment, both the **internal** and the **external consistency** of those models.

Internal consistency relates to the mathematical correctness of the statements in the model being validated. A set of inputs are collected from the system represented by the model, along with relevant information on the assumptions made about the system elements. The model allows computing a set of outputs representing the predicted behavior of the system being modeled. In an internally consistent model, the outputs are valid if the inputs are also valid.

A model exhibits external consistency if information collected from it is not contradicted by other information observed in practice. This relates to the applicability of the model, as it focuses on the extent to which the assumptions made in the model apply beyond the scope of from which the model was delivered.

In our exploratory case study, we noted that the number of methods defined in each of the components is fairly low. It is common to develop toy examples to illustrate techniques, in an academic context. The analyzed metrics are defined as ratios. The small number of elements in their computation may be regarded as a fragility of this heuristics-based quality model. Indeed, the standard deviation of the values for some of the metrics (most noticeably `RCO`) was very high. In contrast, the heuristics were computed with a larger sample of commercial components. Those COTS components were likely to have a higher complexity.

This observation suggests that if the FukaBeans are reusable, in spite of violating some of the reusability heuristics, this may indicate a lack of external consistency of the reusability model proposed by Washizaki *et al.* Of course, further validation of this observation is sought.

Lessons learned

We can organize the discussion of lessons learned in this case study around two main subjects: the formalization of the metrics set, and the limitations of scope of this metrics set which were identified while conducting this case study.

The original metrics definition is ambiguous in what concerns inheritance. It is unclear how inherited features (methods and attributes) should be accounted for. Our formalization only uses the directly defined features. While for this particular sample of components this is not a problematic issue, it is possible to define hierarchies of object-oriented components where this option would have an influence on the metrics values. The shortcoming of the original definition is that we were left with the decision of whether or not we should include inherited methods. Different external validation efforts lead by different research teams could have chosen to use also the indirectly available features. This would severely damage the comparability of results, particularly if their option was not made clear in the experiment report.

The original definitions of the metrics raise another, more subtle issue. Whenever defining the ratio metrics, Washizaki *et al.* specified, in an arbitrary fashion, the metrics results for those components with no accessible properties (in `RCO` and `RCC`) or no business operations (in `SCCp` and `SCCr`). For instance, in the absence of accessible properties, `RCC` equals 1. In other words, the rate of component customizability is maximum. All the 0 (zero) accessible properties are writable. Conversely, one could also say that none of the 0 accessible properties were writable. A similar argument can be made for each of the ratios, although the chosen value in each case varies. This issue could have been better dealt with by clearly specifying that the metrics `RCO` and `RCC` would only be applicable when the pre-condition `A > 0` holds. Similarly, the metrics `SCCp` and `SCCr` would only be applicable when `B > 0` holds. The improved definitions in OCL would be as presented in listing 4.13.

Listing 4.13: Defining pre-conditions for metrics definitions, in OCL.

```
context Component
RCO(): Real = self.Pr()/self.A()
pre: self.A() > 0

RCC(): Real = self.Pw()/self.A()
pre: self.A() > 0

SCCp(): Real = self.Bv()/self.B()
pre: self.B() > 0

SCCr(): Real = self.Bp()/self.B()
pre: self.B() > 0
```

Concerning the limitations of scope of Washizaki *et al.*'s metrics set, the metrics were designed to assess reusability of fine grained components (*JavaBeans*) through

the analysis of their interface complexity. This limits somewhat the scope of model elements being analyzed. UML architectural components have a much richer expressiveness than the one used in these metrics, which leaves out important model elements such as the provided and required interfaces, as well as the events the component may produce or consume.

Another possible concern relates to the complexity associated with parameter types in the evaluation of the complexity of method interfaces. The metrics just count the number of parameters, thus being blind to parameter type repetition and parameter type complexity. For instance, a method with N parameters of distinct types is intuitively more complex than another method with N parameters of the same type. Also, arguments of atomic types (e.g. Integer, Real or Boolean) are intuitively less complex than those of a composed type.

4.4.7 Case study's conclusions and further work

The cross validation of software metrics and quality models is an essential step toward their promotion and subsequent adoption by practitioners. As we have seen in chapter 2, the current state of the art concerning metrics for CBD clearly shows a lack of validation of metrics proposals, not only by their authors, but also, and more importantly, by their peers.

Although it can be argued that most of the proposals are fairly recent, our experience in the area of experimental software engineering lead us to think that there are a few shortcomings hampering the independent collection and cross-validation of most software metrics: either the ambiguity in their definition, when an informal metrics definition technique is used, or the usage of formal definitions using a formalism that is not easy to grasp by practitioners. Furthermore, the lack of availability of adequate tool support for metrics collection is also a common problem.

ODM has helped us overcoming these problems in an elegant and sound way. Using OCL upon the UML 2.0 metamodel, we have metrics definitions which are formally defined, and can be directly used to support the metrics collection, as long as a UML tool with OCL support is used.

Our concern in using standard notations and technologies with a wide adoption by practitioners aims at bridging the traditional gap between research and industry. ODM can be fully integrated in the normal software development process. We regard this as an enabling condition for its widespread adoption.

By facilitating the independent replication of metrics validation efforts, we are providing an essential support for the adoption of the experimental approach advocated in chapter 3. This replication is essential so that independent teams can conduct their own validation efforts, each mitigating its own set of threats to validity. Hopefully, the independent validation efforts will cover, as a whole, the most important threats. In contrast, a validation performed by a single team, even if with replications, is more

prone to repeat validity threats that may result from that team's own biases.

With respect to the results of this independent validation effort, we identified a potential model calibration problem. Our results suggest that the model is not accurate for very fine-grained components. Further differentiated replications should be performed to confirm, or deny, this observation. In particular, it would be interesting to contrast highly reusable components with components that are of a lower quality, so that the heuristics-based quality model proposed by Washizaki *et al.* can be fine tuned, if necessary.

4.5 Related work

4.5.1 ODM applications to other domains

Object oriented design metrics

These metrics fall in the category of metrics covered by M2DM. M2DM was originally proposed to support the specification and collection of metrics for object-oriented design [Abreu 01b, Abreu 01a] and was used in the formalization of several object-oriented design metrics suites [Baroni 02b, Baroni 03, Baroni 02a]. Although the first metrics formalizations used a metamodel that was based on a research object-oriented design language called GOODLY [Abreu 99], the approach was subsequently ported with success to the UML 1.* metamodel.

M2DM was recently adopted in industry by a UML tool producer[11] to add the capability of metrics collection to a UML tool. The tool uses a simplified version of a subset of the UML 2.0 metamodel as a base metamodel upon which OCL expressions are used to support metrics collection.

There are a few proposals by other authors which can be compared to the M2DM approach (and, as such, to ODM). In [Moser 97], Moser and Misic proposed a formal metamodel approach to measuring class coupling and cohesion. They used the Z formal notation to define the ontology and the metrics themselves. Their approach is elegant and sound, but the usage of Z is, in our opinion, a drawback with respect to the knowledge transfer to industry, as most practitioners are not comfortable with formal languages. In [Morisio 99], Morisio proposed metrics using OMT scripts upon process models. To make them easier to understand, he translated the definitions also to SQL. Again the main rationale behind ODM applies, although in this case with slightly less formality than our approach, or Moser's.

[11] Borland, (`http://www.borland.com/`), with its Together Architect 2006. The tool uses OCL defined metrics to support heuristic assessment on models. The base metrics set is open, so that users may add new metrics and their corresponding heuristics.

Object-relational database metrics

The usage of different metamodels for the collection of object-oriented design metrics is an instance of a more general property of M2DM (and, consequently, of ODM). The approach is generic in what concerns the domain of application. It has been used to define, collect, and use metrics on other domains, such as that of relational databases [Baroni 05b, Baroni 05a, Calero 05].

Component-based development metrics

In the scope of the work described this dissertation, we use ODM in essentially two different contexts: for process metrics, collected during the development of software components, and for metrics collected on existing component-based systems. The former usage will be presented in detail on chapter 6.

The latter was used in metrics formalization and collection for different types of component technologies (each corresponding to a specific metamodel). We collected metrics on JavaBeans, using the UML 2.0 metamodel, as shown in this chapter, and also on [Goulão 04b, Goulão 04c, Goulão 04a, Goulão 05c]. Corba Components, using the CCM metamodel [Goulão 05b] and component assemblies, using an extended version of the CCM metamodel [Goulão 05a], were also used to explore the versatility of the ODM approach, as we will discuss in chapter 5. Finally, we will use ODM on Eclipse plug-ins, using a metamodel developed for this purpose as the ontology, in chapter 7.

4.6 Conclusions

In this chapter, we presented a metrics formalization and collection technique called Ontology-Driven Measurement (ODM). We exemplified how metrics can be formally defined upon an ontology (in this case, a metamodel), and how they can be computed. Then, we presented an exploratory case study concerning the independent validation of a metrics set and its underlying quality model, proposed by Washizaki *et al.* for assessing the reusability of JavaBeans. We tested a set of components developed by Washizaki's team, for educational purposes, against the set of heuristics proposed in their quality model. The results suggest that the heuristics-based quality model may require some calibration, when used with fine-grained components. Perhaps more importantly, at least for the purposes of this dissertation, this exploratory case study was used as a pretext for exercising ODM in the context of software component design analysis. The techniques introduced in this chapter will be further explored in the following chapters.

Chapter 5

ODM expressiveness assessment

Contents

Background: *If we are to adopt ODM as a metrics definition technique for CBD, it is important to explore the expressiveness of this approach in the context of CBD.*

Objective: *Our goal is to assess the expressiveness of component metamodels in metrics formalization, illustrating both the flexibility of ODM and how some common difficulties relating to that expressiveness can be circumvented.*

Method: *We present the definitions of metrics sets, proposed by several authors, to cover different aspects of component interaction. We formalize those metrics using the UML 2.0 and CCM 3.0 metamodels. We assess the formalization exercise.*

Results: *Metrics formalization facilitates the identification of shortcomings of the original metrics definitions. We also identify some limitations in the UML 2.0 and CCM 3.0 metamodels, and propose metamodel extensions to solve them.*

Limitations: *There are other component models, each with its own constructs, which could have been used in this formalization exercise. A possible extension to this work would be to repeat this formalization with such models.*

Conclusions: *The formal definition of metrics exposes several shortcomings of their "traditional" definitions, and illustrates the expressiveness and flexibility of the ODM approach using different metamodels.*

5.1 Introduction

In the previous chapter we introduced ODM as a rigorous approach to the specification and computation of software metrics. We illustrated the ODM approach by specifying and collecting metrics defined on the interfaces of JavaBeans. The metrics were formalized using OCL expressions upon the UML 2.0 metamodel. The metrics set used in chapter 4 is a typical example of a set of product metrics defined upon software artifacts (in chapter 4's case, class diagrams obtained by reverse-engineering the source code of the JavaBeans).

In this chapter we will explore the expressiveness of the ODM approach in the context of CBD. While on the previous chapter the UML 2.0 was used as a metamodel for metrics definition, in this chapter we will show how different metamodels can be used to specify and collect the same metrics.

The focus of this chapter is on the expressiveness of the ODM approach and its suitability to metrics definition in different contexts, rather than on the the usefulness of each of the defined metrics. Such usefulness can only be assessed through experimentation. Experimentation in CBD will be the focus of chapters 6 and 7.

This chapter is organized as follows: we start by introducing a toy example using both the UML 2.0 and the CCM 3.0 as the specification notations. Then, we informally describe a set of metrics for software components, which includes metrics proposed by several authors. Our presentation of these metrics uses their original specification notations, and highlights the shortcomings of such notations. We use the toy model to illustrate the computation of the presented metrics. The metrics are then formalized using the ODM approach, both upon the UML 2.0 metamodel and the CCM 3.0 metamodel. The limitations of these metamodels for the purposes of our formalization effort are discussed, as well as proposals to solve the identified problems. The chapter ends with a discussion on the evolution of the complexity of metamodels and its impact on the usability of the ODM approach.

5.2 A component assembly toy example

Consider a car company that is specifying the information display system for several car models. The company plans to use a wide range of features from the basic speed, engine rotation, and temperature indicators to parking cameras, a GPS system, and so on. Our example will cover three different models, from the low to the high end version of a information display system.

5.2.1 Structural model in UML 2.0

Our example will be presented in UML using component diagrams, one for each of the three component assemblies that correspond to the three car models. Before describing

in some detail the different component assemblies, we start by a quick remark on notation. In these component diagrams, some interfaces are decorated with the `<<event>>` stereotype, to represent produced and consumed events. So, for example, in component assembly **A** (5.1), `Clock` produces a `TimerTick` event which is then consumed by the `LED_Display` and the `Engine` components. Other specification formalisms, such as the CCM have specific model elements for expressing this concept, but that is not the case with UML 2.0.

Figures 5.1 through 5.3 present the configurations used, from the lower-end car model, to the higher-end one.

Assembly **A** (figure 5.1) uses a LED (Light-Emitting Diode) display, with the typical instruments of a car: a velocimeter, a fuel gauge, a coolant temperature gauge, engine warning lights, and a clock. To keep our example small and simple we will assume that:

- there is a `Clock` component providing all the time information required by the assembly;
- the `Engine` component wraps all the car engine diagnosis instruments;
- these components neither provide, nor require, any additional interfaces, other than those presented here.

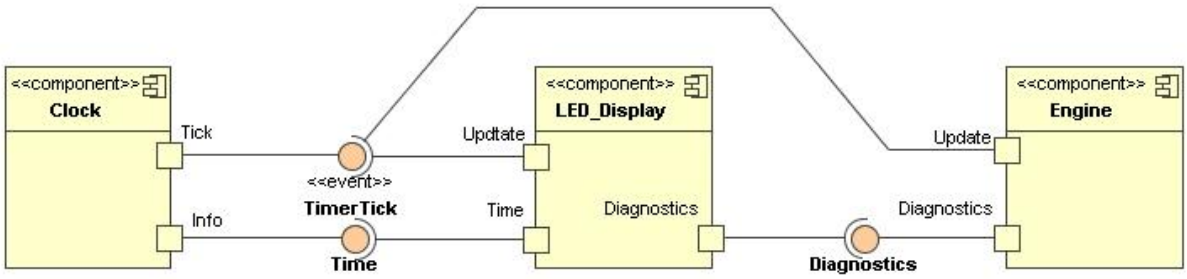


Figure 5.1: Low-end car model configuration (assembly A)

Assembly **B** (figure 5.2) is similar to assembly **A**, except for the display component. Rather than a LED-based instrument panel, assembly **B** uses a colored TFT LCD(Thin Film Transistor Liquid Crystal Display) video display. All the information made available to the LED-based instrument panel can also be displayed with the video display. However, the video display has a higher flexibility with respect to the information it can present. It can be connected to a GPS (Global Positioning System) device, through its required `Location` interface, to an external device, such as a DVD (Digital Versatile Disc) player, through the `ExternalDevice` interface, to a Terra Trip device (a device often used in vehicles to allow a detailed registration of distances traveled by the vehicle during all-terrain competitions), through the `TerraTrip` interface, and to a parking camera (used to help parking the car, through the `Camera` interface. In assembly **B**,

although the video display supports all the above mentioned interfaces, it keeps on using only the same sources of information as in assembly **A**.

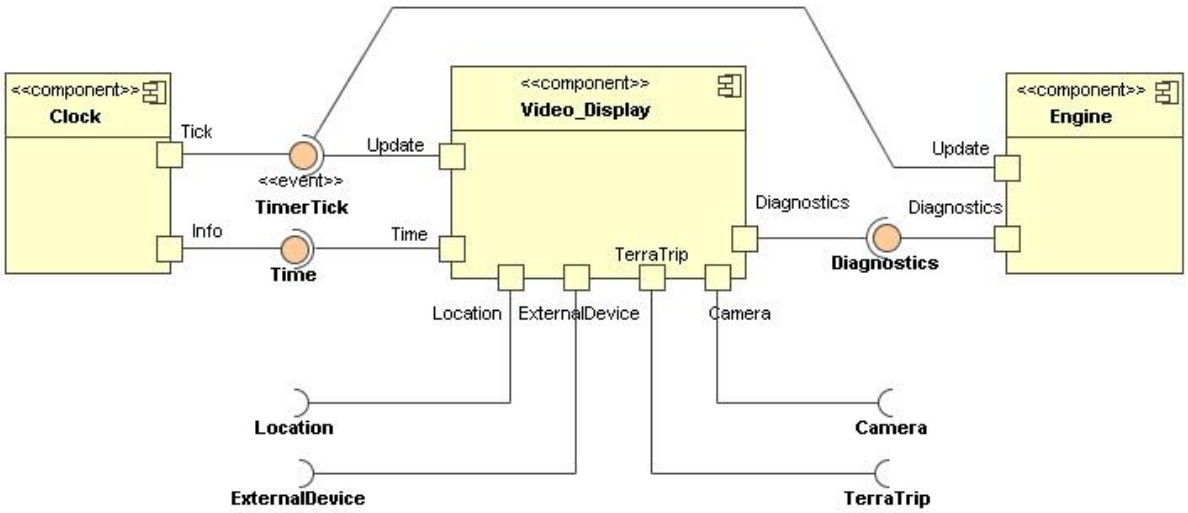


Figure 5.2: Middle-range car model configuration (assembly B)

Assembly **C** (figure 5.3) keeps the characteristics of assembly **B**, and adds to them the usage of some of the devices which can be plugged in the `Video_Display` component. In particular, it integrates a GPS device, as well as a parking camera.

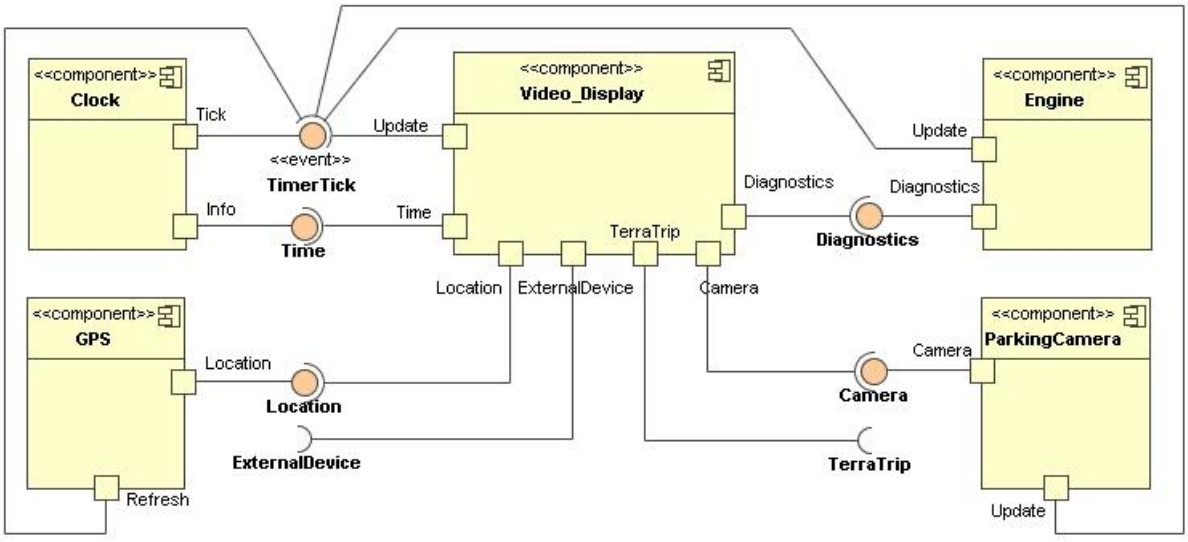


Figure 5.3: High-end car model configuration (assembly C)

Figure 5.4 presents the interfaces used in our car model examples. The details of these interfaces, concerning how adequate they would be in reality, are not particularly relevant for our discussion, as we are only interested in measuring some properties of those interfaces.

The `TimerTick` interface only has one method, and is used for the sake of the `Clock` component firing a timing event to be consumed by other components. The `Time` interface has a set of typical operations on a data type representing time in terms of

year, month, day, hours, minutes and seconds. The `Diagnostics` interface condenses the information made available by the car engine, such as current speed, engine rotations, oil level, temperature and remaining fuel. The `Location` interface is used for geo-referential information. The `Camera` interface is used for video streaming, so that a parking camera can be connected to the vehicle and the images captured by the camera can be displayed to the driver. The `ExternalDevice` interface extends the basic `Camera` interface with the operations required for video playback. Finally, the `TerraTrip` interface allows the connection of a Terra Trip device.

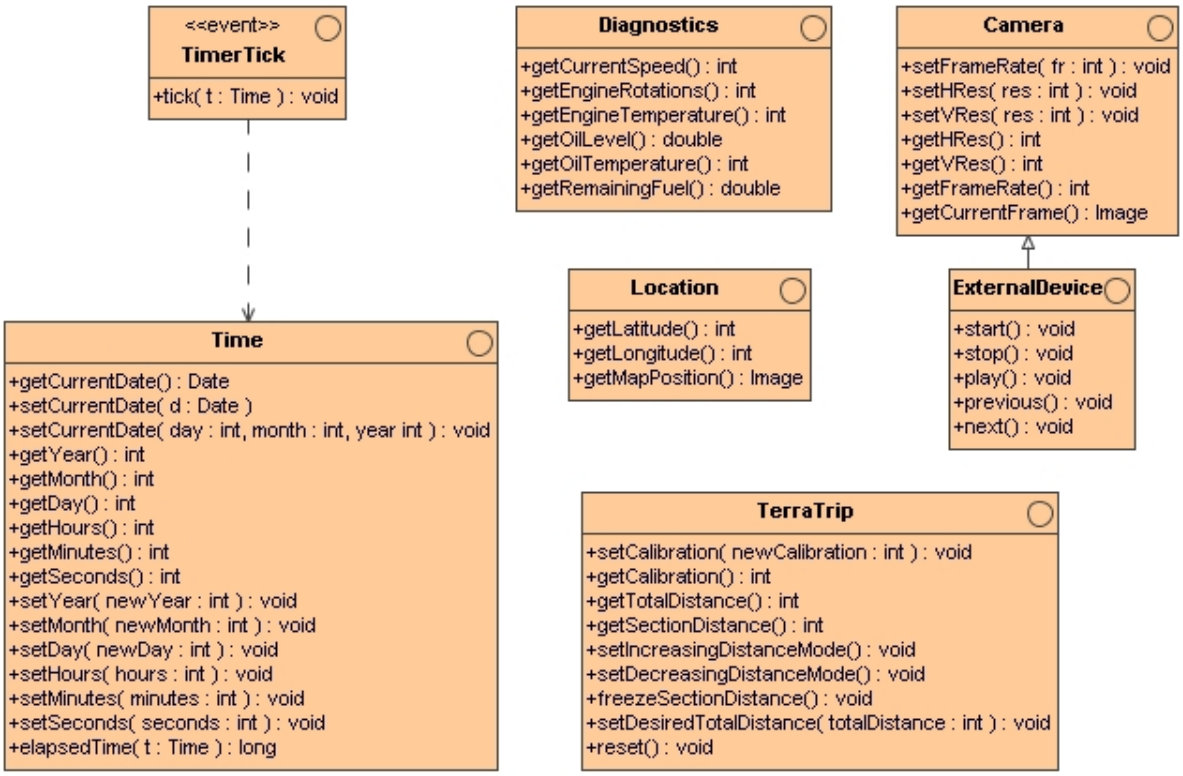


Figure 5.4: Interfaces used in our car example

5.2.2 Structural model, in CCM

Figures 5.5 through 5.7 present the component assemblies of our three different car models. With the exception of a few notation details, when compared to their UML 2.0 counterparts, they are essentially similar. Note that, unlike what happens with UML 2.0, the CCM has a specific notation for representing events in these diagrams.

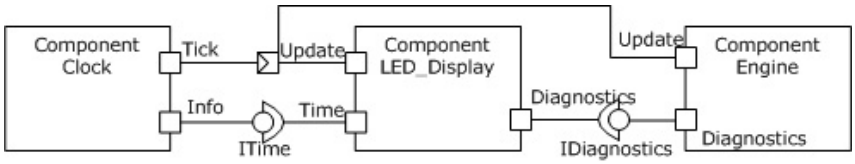


Figure 5.5: Low-end car model assembly, in CCM

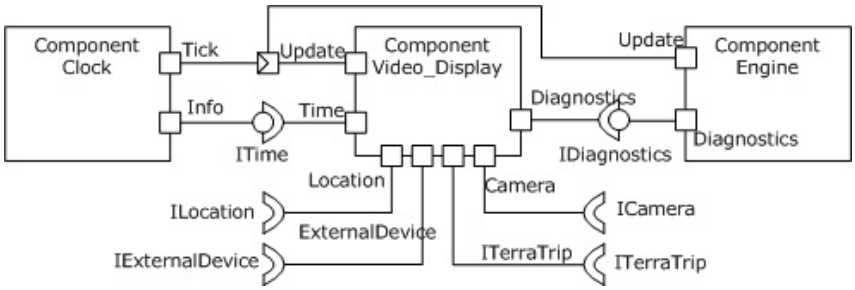


Figure 5.6: Middle-end car model assembly, in CCM

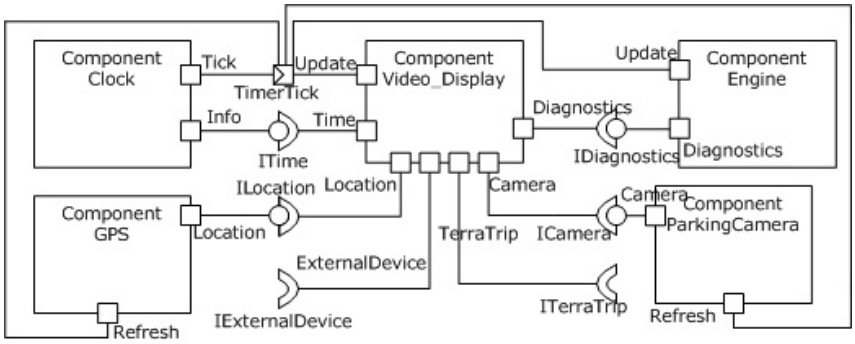


Figure 5.7: High-end car model assembly, in CCM

In this component model, the interfaces are defined using a **C**ommon **I**nterface **D**escription **L**anguage (CIDL). Listing 5.1 presents the definition of the `ICamera` interface, for illustration purposes. All these interfaces are essentially similar, in what concerns their signature, to those presented earlier for this example in UML 2.0 (see figure 5.4). As with the UML 2.0 assembly description, the existence of an `Image` type is assumed, rather than modeled here.

Listing 5.1: ICamera interface, in CIDL.

```
interface ICamera {
  void setFrameRate(in int fr);
  void setHRes(in int res);
  void setVRes(in int res);
  int getFrameRate();
  int getHRes();
  int getFrameRate();
  Image getCurrentFrame();
}
```

5.2.3 Concerns addressed in our example

In this example, we address two concerns:

- In component-based development, available components often do not exactly match the requirements they are to fulfill, either because they lack functionality, or because they have functionalities that are not used by the assembly they are put into. The question we will try to answer is how well do the components fit into the component assembly, with respect to the actual usage of their interfaces.

- The complexity of interfaces, either provided or required, varies significantly. This may have an impact in the usability of components, from the component integrator's point of view. An over-simplistic interface may be easy to understand and use, but lacks the flexibility for allowing a high level of customizability of the component. Conversely, a complex interface may provide that flexibility at the expense of the interface's understandability.

The architectural mismatches between available and required resources may result in two kinds of problems:

- The functionalities required by the components will not be fully available. In our example, this is mostly noticeable in assembly B, where the video display supports several other devices, which are not present in this assembly.

- Some functionalities provided by the components in this assembly are not used. If these components were not to be reused in other assemblies, this would imply that effort would have been spent in the developed of unused functionalities. In our examples, all provided interfaces are required by at least one component.

With respect to the complexity of the interfaces, we deliberately kept them fairly simple, to facilitate comparison. We used an inconsistent naming policy in one of the interfaces (`Time`) by using different names for, essentially, the same arguments. As such, `day`, `month`, and `year`, in the `setCurrentDate` method correspond to `newDay`, `newMonth`, and `newYear`, in `setDay`, `setMonth`, and `setYear`, respectively. This will allow us to explore how one of the metrics used in this chapter can help developers to detect such inconsistencies.

5.3 Informal description of structural metrics

In this section, we briefly present the product metrics to be formalized using different component metamodels. For the sake of uniformity, we follow a similar pattern for each metric, or group of related metrics. We start by presenting their name and original specification, keeping the notation used by their proponents, thus illustrating the variability of notations commonly used in metrics definitions. Then, we present the metric's rationale, in their proponents' view. These are followed by considerations and assumptions that will be made during the formalization process of the metric in OCL,

whenever there is information which is missing, or unclear, in the metrics' informal specifications.

We start by presenting component metrics which are collected on individual components, regardless of their usage in any given component assembly. These include proposals by Boxall and Araban [Boxall 04], Narasimhan and Hendradjaya [Narasimhan 04], and Goulão and Brito e Abreu [Goulão 05b].

We also present proposals to assess the fitness of components in a specific architecture, by Hoek *et al.* [Hoek 03], and Narasimhan and Hendradjaya [Narasimhan 04]. These metrics are context specific, in the sense that the same component will have a different metric value depending on the assembly where it is being integrated. While some of the context specific metrics are aimed at individual components, with respect to their integration in the assembly, others are measured for the whole component assembly.

5.3.1 Component metrics

Component interface complexity assessment [Boxall 04]

The metrics presented in this section aim to assess the understandability of a component interface. Boxall and Araban assume that understandability has a positive influence on the reusability of software components. As such, a higher understandability leads to a higher reusability of the components.

Original specification:

Arguments Per Procedure (**APP**) represents the average number of arguments in publicly declared procedures (within the interface), and is defined in Eq. 5.1.

$$APP = \frac{n_a}{n_p} \tag{5.1}$$

where:
n_a = number of arguments on the publicly declared procedures
n_p = number of publicly declared procedures

The **Distinct Argument Count**(**DAC**) represents the number of distinct arguments in publicly declared procedures and is defined in Eq. 5.2. The **Distinct Arguments Ratio** (**DAR**) represents **DAC**'s percentage on the component interface and is defined in (Eq. 5.3).

$$DAC = |A| \tag{5.2}$$

where:
A = set of the $< name, type >$ pairs representing arguments in the publicly declared procedures
$|A|$ = cardinality of the A set.

$$DAR = \frac{DAC}{n_a} \tag{5.3}$$

The **Arguments Repetitiveness Scale** (**ARS**) aims to account for the repetitiveness of arguments in a component's interface (Eq. 5.4). In other words, it is used for measuring the consistency of the naming and typing of arguments within the publicly declared procedures of an operation.

$$ARS = \frac{\sum_{a \in A} |a|^2}{n_a} \tag{5.4}$$

where:
$|a|$ = count of procedures in which argument name-type a is used in the interface

Rationale:

The rationale for the **DAC** metric is that humans have a limited capacity of receiving, processing, and remembering information [Miller 56]. The number of information chunks in the procedure definition (in this case, its arguments) should be limited. Boxall and Araban suggest that an increased number of arguments reduces the interface's understandability and, therefore, its reusability.

The rationale for the **DAR** metric is that the repetitiveness of arguments increases the interface's understandability, and, therefore, the component's reusability. According to Boxall and Araban, $|a|$ is squared in this definition to create a bias that favors consistent arguments definitions in the interface. Boxall and Araban also claim that interfaces with a higher **ARS** *"will tend to be dominated by fewer distinct arguments which are repeated more often"*.

DAC is influenced by the adoption of a consistent naming convention for arguments in the operations provided by a component. If the same argument is passed over and over to the component's operations, the effort required for understanding it for the first time is saved in that argument's repetitions throughout the interface. The smaller the number of distinct arguments a component user has to understand, the better. Likewise, a lower **DAR** leads to a higher understandability. However, unlike **DAC**, **DAR** is immune to the size of the interface, because its value corresponds to **DAC**, when normalized by n_a.

Comments:

The original proposal of the **APP** metric uses C/C++ component interfaces to illustrate the metric definition. Overloaded and overridden operations are referred to in the definition, but not inherited ones. We assume them to be outside the scope of this metric. If the component is implemented in an OO language, all public and protected OO methods should be counted, but not the private ones, as these will be invisible to component users. **APP**'s definition assumes a single, or at least unified, interface for the component. As seen in our example, components may provide or require more than one interface. As each interface has a set of operations, we can consider the "unified interface" as the result of the union of the interfaces. We will also assume that the operation names will be qualified by the interfaces they belong to, so that operations with a similar signature owned by different interfaces are counted as different operations, rather than the same.

In the **DAC** metric definition, Boxall and Araban consider a parameter as a duplicate of another if the pair $< name, type >$ is repeated in different operation signatures. The same holds for the definitions of **DAR** and **ARS**.

Component internal complexity [Narasimhan 04]

Original specification:

The **Component Packing Density** (**CPD**) was proposed by Narasimhan and Hendradjaya and aims at assessing the complexity of a component, with respect to the usage of a given mechanism. **CPD** represents the average number of constituents of a given type (e.g. lines of code, interfaces, classes, modules) in a component (Eq. 5.5).

$$CPD_{constituent\_type} = \frac{\sharp < constituent\_type >}{\sharp components} \tag{5.5}$$

where:
$constituent\_type$ = type of the constituents whose density is being assessed
$\sharp < constituent\_type >$ = number of elements of $constituent\_type$ in the assembly
$\sharp components$ = number of components in the assembly

Rationale:

A higher density indicates a higher complexity of the component, thus requiring, as Narasimhan and Hendradjaya suggested, a more thorough impact analysis and risk assessment.

Comments:

CPD can be defined for a multitude of different constituents, but most of those suggested by Narasimhan and Hendradjaya are not available for users of black-box components. For illustration purposes only, we will assume the operations in all pro-

vided interfaces as the constituents of a component, in our formalization.

Component communication [Goulão 05b]

The metrics presented so far in this chapter focus mainly on the provided interfaces of components, and the operations defined there. In what concerns their definition, adapting these metrics to the required interfaces of components would be straightforward. The ODM can also be used on other items of a component interface, such as events. Although there is no direct support for representing produced and consumed events in the UML 2.0, these can be suitably represented in other notations, such as the CCM. Furthermore, the UML lightweight extension mechanism (stereotypes) can be used to circumvent this limitation of the standard metamodel.

Original specification:

We can define **Event Fan-In** (**EFI**) to measure the number of provided events. In some metamodels, such as the CCM metamodel, these events are either emitted or published by a component [1]. **Event Fan-Out** (**EFO**) represents the number of events consumed by the component.

The paper in which we proposed both metrics [Goulão 05b] was dedicated to metrics defined upon CCM. `EFI` and `EFO` are defined as operations of the `ComponentDef` meta-class. The definitions rely on auxiliary operations `PublishesCount()`, `EmitsCount()`, and `ConsumesCount()`, also defined upon the same meta-class. For this initial presentation of the metrics, we will omit the auxiliary operations, and just leave the metrics definitions. Later, when presenting the formal definitions for all the metrics presented in this chapter, we will revisit these definitions and complete them with all the necessary auxiliary operations. Listing 5.2 presents the defintions of `EFI()` and `EFO()`.

Listing 5.2: The EFI and EFO metrics.

```
context ComponentDef
  EFI(): Integer = self.PublishesCount() + self.EmitsCount()
  EFO(): Integer = self.ConsumesCount()
```

Rationale:

For **EFI** and **EFO**, the understandability of the component interaction with other components gets lower as the number of events gets higher. In other words, a higher complexity leads to a lower understandability.

[1] The distinction between emitted and published events is not present in all component models. This distinction is used in the CCM, for instance, but not in our extension of the UML 2.0 metamodel, where we will simply consider events as either provided or required (just like interfaces). A publisher component is an exclusive provider of an event, while an emitter shares an event channel with other event sources. In any case, components may subscribe to that event channel, to receive the events notifications.

Comments:

Events are not supported by all the component models currently available. Therefore, these metrics will not be available for component models that do not support their underlying model elements, either directly (as it happens with the CCM), or indirectly (as in UML 2.0, through the usage of stereotypes).

5.3.2 Assembly-dependent component metrics

Component service utilization metrics [Hoek 03]

Original specification:

The **Provided Services Utilization** (**PSU**) represents the ratio of services provided by the component which are actually used (Eq. 5.6).

$$PSU_X = \frac{P_{actual}}{P_{total}} \tag{5.6}$$

where:
P_{actual} = number of services provided by component X that are actually used by other components
P_{total} = number of services provided by component X

The **Required Services Utilization** (**RSU**) is similar, but for required services (5.7).

$$RSU_X = \frac{R_{actual}}{R_{total}} \tag{5.7}$$

where:
R_{actual} = number of services required by component X that are actually provided by the assembly
R_{total} = number of services required by component X

Both **PSU** and **RSU** are measured for each component. The **Compound Provided Service Utilization** (**CPSU**) and the **Compound Required Service Utilization** (**CSRU**) metrics can be informally defined as the ratio of services provided by components that are actually used by the component assembly (Eq. 5.8), and the ratio of services required by the components that are actually provided by the component assembly (Eq. 5.9), respectively.

$$CPSU = \frac{\sum_{i=1}^{n} P^i_{actual}}{\sum_{i=1}^{n} P^i_{total}} \tag{5.8}$$

where:
P^i_{actual} = number of services provided by component i that are actually used by other components
P^i_{total} = number of services provided by component i

$$CRSU = \frac{\sum_{i=1}^{n} R^i_{actual}}{\sum_{i=1}^{n} R^i_{total}} \tag{5.9}$$

where:
R^i_{actual} = number of services required by component i that are actually provided by the assembly
R^i_{total} = number of services required by component i

Rationale:

PSU denotes the extent to which the assembly uses the services provided by the component. A low value of **PSU** may occur if a component was built for reuse in several contexts, thus providing several services for component reusers to choose from. The downside is that this also means that the component carries a large amount of extra functionality that is not required by the assembly.

RSU denotes the extent to which a component requires services that are available in the component assembly. Ideally, **RSU**'s value should be 1, meaning that all the required services are available, but this is not always the case. The unavailability of required services may impact the component in different ways, from partial loss of functionality, or performance, to rendering the component useless in this assembly. This impact may also have repercussions in the rest of the assembly: the loss of functionality on the directly impacted component may affect the components that have dependencies on those functionalities, and so on.

As noted by Wallnau and Stafford [Wallnau 02], component assemblers are usually more interested in the overall properties of the component assembly than on the individual properties of each of the used components. **CPSU** and **CPRU** can be used to help assessing the properties of the component assemblies.

Comments:

Hoek *et al.*'s definitions are generic, in the sense that they deliberately do not prescribe how to instantiate the notion of service. The notion of service, as presented by Hoek *et al.*, covers any kind of publicly accessible resource of the component, such as operations and data structures. The granularity of what is considered a service may also be fine tuned, according to which is more suitable for the assessor's purpose and the expressiveness of the Architecture Description Language used in the component specification.

In this chapter, we assume that the services correspond to the interfaces provided and required through the component's communication ports. This is a plausible assumption, as we are considering interfaces as the *"reuse plugs"* of components.

Note that we could also assume that each service corresponds to an operation specified in those interfaces, rather than to the interfaces. We would still be implementing Hoek *et al.*'s metrics, but the results would not be comparable to the ones obtained here.

Interaction density of a component [Narasimhan 04]

Original specification:

The **Interaction Density of a Component** (**IDC**) is defined as a ratio of actual interactions over potential ones (Eq. 5.10). The **Incoming** and **Outgoing Interaction Density of a Component** (**IIDC** and **OIDC**, respectively) are similar, but consider only incoming interactions (Eq. 5.11) or outgoing ones (Eq. 5.12).

$$IDC = \frac{\sharp I}{\sharp I_{max}} \tag{5.10}$$

where:
$\sharp I$ = Actual Interactions
$\sharp I_{max}$ = Maximum available interactions

$$IIDC = \frac{\sharp I_{IN}}{\sharp I_{max_{IN}}} \tag{5.11}$$

where:
$\sharp I_{IN}$ = Actual incoming interactions
$\sharp I_{max_{IN}}$ = Maximum available incoming interactions

$$OIDC = \frac{\sharp I_{OUT}}{\sharp I_{max_{OUT}}} \tag{5.12}$$

where:
$\sharp I_{OUT}$ = Actual outgoing interactions
$\sharp I_{max_{OUT}}$ = Maximum available outgoing interactions

The **Average Interaction Density of Software Components** (**AIDC**) represents the sum of **IDC** for each component divided by the number of components (Eq. 5.13).

$$AIDC = \frac{IDC_1 + IDC_2 + \ldots + IDC_N}{\sharp components} \tag{5.13}$$

where:
IDC_i = IDC of component i
$\sharp components = N$ = number of components in the system

Rationale:

A higher interaction density causes a higher complexity in the interaction. Narashiman and Hendrajaya regard this complexity as a source of risk that should be taken into account when assigning professionals to component design. The rationale would be to assign the most experienced developers to denser interactions.

Comment:

IDC is somewhat similar to a combination of **PSU** with **RSU** but Narasimhan and Hendradjaya consider the usage of both interfaces and events as interactions. The parallel can also be made for **IIDC** with **RSU**, and **OIDC** with **PSU**.

5.3.3 Collected metrics

The metrics described in the previous sub-section can be collected upon our toy example. We start by presenting in table 5.1 the metrics which are collected upon components, and are defined in such a way that their value depends on the component's characteristics, regardless of the component integration into component assemblies. The first column presents the component assembly - A, B, or C - to identify the context under which each component is being measured. NA represents the values which could not be computed (e.g. divisions by 0).

Assembly	Component	APP	DAC	DAR	ARS	CPD	EFI	EFO
A	**Clock**	0,69	11	1,00	1,00	16	1	0
	LEDDisplay	NA	0	NA	NA	0	0	1
	Engine	0,00	0	NA	NA	6	0	1
B	**Clock**	0,69	11	1,00	1,00	16	1	0
	VideoDisplay	NA	0	NA	NA	0	0	1
	Engine	0,00	0	NA	NA	6	0	1
C	**Clock**	0,69	11	1,00	1,00	16	1	0
	VideoDisplay	NA	0	NA	NA	0	0	1
	Engine	0,00	0	NA	NA	6	0	1
	GPS	0,00	0	NA	NA	3	0	1
	ParkingCamera	0,43	2	0,67	1,67	7	0	1

Table 5.1: Component metrics

Table 5.2 presents the metrics collected on each component instance, for each of the

assemblies. The metrics presented in this table are defined so that their value depends on the component assembly in which the component is integrated.

Assembly	Component	PSU	RSU	IDC	IIDC	OIDC
A	**Clock**	1,00	NA	1,00	NA	1,00
	LEDDisplay	NA	1,00	1,00	1,00	NA
	Engine	1,00	NA	1,00	1,00	1,00
B	**Clock**	1,00	NA	1,00	NA	1,00
	VideoDisplay	NA	0,33	0,43	0,43	NA
	Engine	1,00	NA	1,00	1,00	1,00
C	**Clock**	1,00	NA	1,00	NA	1,00
	VideoDisplay	NA	0,67	0,71	0,71	NA
	Engine	1,00	NA	1,00	1,00	1,00
	GPS	1,00	NA	1,00	1,00	1,00
	ParkingCamera	1,00	NA	1,00	1,00	1,00

Table 5.2: Component metrics

Finally, table 5.3 presents the metrics collected on each of the component assemblies.

Assembly	CPD	CPSU	CRSU	AIDC
A	7,33	1,00	1,00	1,00
B	7,33	1,00	0,33	0,81
C	6,40	1,00	0,67	0,94

Table 5.3: Component assembly metrics

5.3.4 Comments on metrics values

The values presented in the previous section illustrate some of the ideas that lead to the proposal of the corresponding metrics. An observation that crosscuts both tables 5.1 and 5.2 is that there are several metrics which could not be computed for some of the components. This highlights a common shortcoming of most informal metrics definitions. It is unclear what to do in the presence of the cases where it is not possible to compute the metric value using the available definition.

While some of the metrics are, essentially, the size of a collection of model elements that share some common property, and therefore are represented by integer values, others are average values, or percentages, and are represented by real values.

Environment free component metrics (Table 5.1)

For each component, the metrics' values in table 5.1 remain the same, regardless of how that component is used in each of the component assemblies. In other words, these metrics are only dependent on the component upon which they are measured, regardless of the context in which this component is used. So, they are examples of the environment-free component metrics referred to in section 2.5.5.

Concerning the metrics values, clearly the `Clock` component has a higher **APP** than the remaining components, which is an indication of a potentially more complex interface, according to this metric.

Only two of the components (`Clock`, and `ParkingCamera`) have arguments in their interfaces. However, as denoted by the values of **DAR** and **ARS**, there is a more frequent repetition of arguments in the `ParkingCamera` component provided interface, which is expected to make it easier to understand, when compared to `Clock`'s provided interfaces. In both components, only one argument gets to be repeated in different operations (`t: Time`, for the `Clock` component, and `res: int`, for the `ParkingCamera` component). However, its relative weight on the overall count of distinct arguments in the component (**DAC**) is heavier in the `ParkingCamera` component.

With respect to **CPD**, the most noticeable components are the `LEDDisplay` and `VideoDisplay`. Neither of them provides any interfaces. As such, because we have instantiated the concept of **CPD** with operations in the provided interfaces, these components have a **CPD** of 0.

The **EFI** and **EFO** metrics in the assembly are related to the `TimerTick` event. As this is the only modeled event, in these assemblies, the components producing it have an `EFI` of 1, while the components consuming it have an `EFO` of 1, respectively. Conversely, the producers of this event have an `EFO` of 0, and the consumers have an `EFI` of 0.

Context dependent component metrics (Table 5.2)

In model **A** there is a perfect match among the interacting components. All the provided services and emitted events are used or consumed by components within the assembly. In the component assembly of model **B**, the `VideoDisplay` component has several required interfaces which are not available within the assembly. In this case, we can suppose that this "waste" of resources occurs mainly because this component was built for reuse. Without additional information, we would prefer model **A** to model **B**, if our priority is to have a simpler component that is still able to fulfill our requirements. On the other hand, if we plan to reuse the assembly by adding other components, model **B** could be a good option, when compared to model **A**. In terms of effectiveness of reuse, model **C** is better than **B**. The comparison of the **RSU** values for the display components can be used to help choosing among different alternatives. These metrics fall into the category of context-dependent component metrics, as discussed in section 2.5.6.

Component assembly metrics (Table 5.3)

Component assembly metrics use the whole component assembly as context, rather than a component. While **CPD** is an average value, the remaining metrics are ratios, varying from 0,00 to 1,00. On the average, the **CPD** is lower in assembly **C** than in assemblies **A** and **B**, which means that the average complexity of the components in

assembly **C** is lower, according to this metric. The **CPSU** has the same value in all assemblies, which means that, for these examples it has no discriminatory power. This would not be the case in assemblies where some of the provided services are not used by any of the components. In contrast, the values of **CRSU** and **AIDC** highlight the fact that, in assemblies **B** (to a greater extent) and **C** (to a lower one), we are not using some components to their full potential. Some resources, which would be required so that the whole functionality of components could be used, are not available in the component assembly. For instance, although the `VideoDisplay` component requires a `TerraTrip` device, that device is not available in any of the assemblies. The perfect match between provided and required resources in assembly A is also noticeable, in metrics **CRSU** and **AIDC**.

5.4 Metrics definition formalization

Each of the metrics informally defined in the previous sections will now be formalized, using the ODM approach upon two different component models: UML 2.0 and CCM 3.0.

The structure of the presentation will be as follows: For each component model, we will briefly discuss its underlying metamodel and present the most relevant metaclasses for our formalization. Then, we will present the metrics formalization in OCL.

5.4.1 UML 2.0

Context specification

The context used in the specification of the metrics is provided by the UML 2.0 metamodel, where all the entities which are relevant for the measurement collection, as well as the relationships among those entities are expressed. We will focus on a portion of the UML 2.0 metamodel that includes the metaclasses and meta-associations which are relevant for the specification and collection of the metrics defined in section 5.3. Figure 5.8 represents a filtered view on the UML 2.0 metamodel, where the metamodel elements which are not relevant for this metrics formalization are omitted.

In this portion of the UML 2.0 metamodel we can observe how a component is represented through the metaclass `Component`. Since the `Component` is an `EncapsulatedClassifier`, it may own ports. Each `Port` may contain an arbitrary number of provided and required interfaces (`Interface`). Provided and required interfaces are wired through assembly connectors (`Connector`, defined in the `BasicComponents` package). These connectors allow specifying the involved ports in their connector ends.

Each `Interface` has a set of owned operations (`Operation`). An operation has a set of parameters. `Parameter` is a sub-class of `TypedElement`, which, in turn, is a subclass

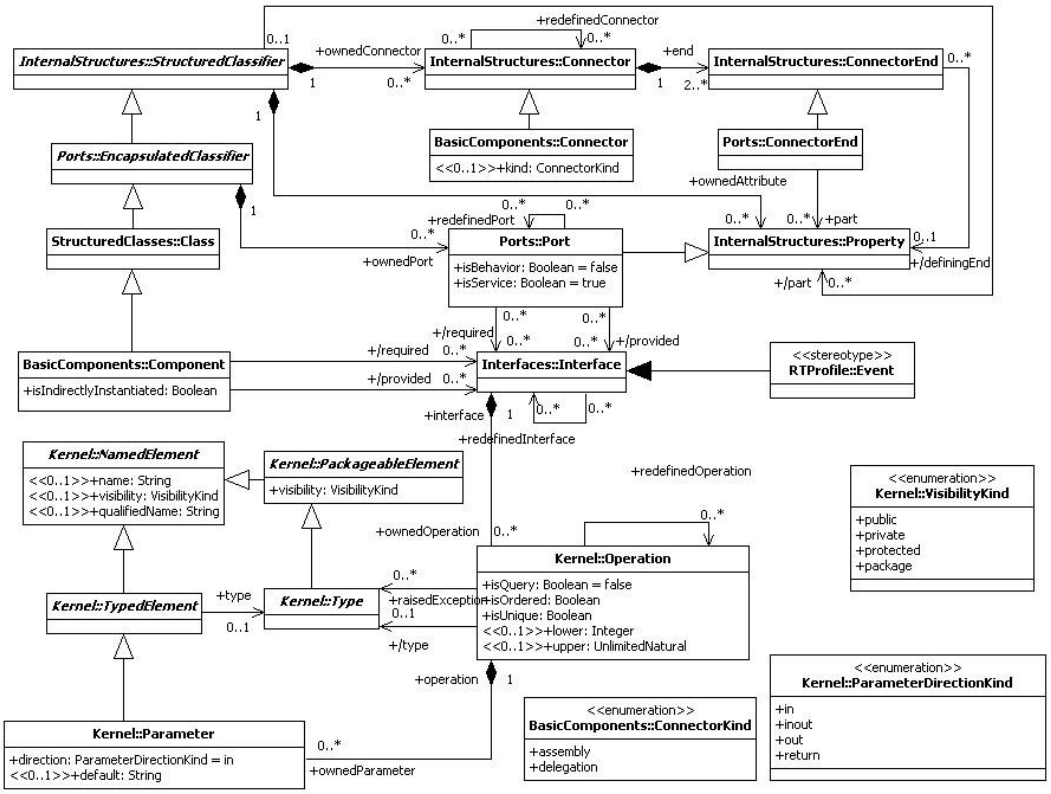


Figure 5.8: A filtered view of the UML 2.0 metamodel, adapted from [OMG 05b].

of `NamedElement`, so each parameter has a name and a type.

Metamodel extension

The UML 2.0 metamodel has no standard way of representing events such as `TimerTick` in a differentiated way, with respect to other interfaces. However, we can model events using UML's lightweight extension mechanisms, by stereotyping the appropriate metaclass. Therefore, we extend the UML metamodel with a new stereotype `<<Event>>`, which is applied to the metaclass `Interface`. The stereotyped interface is represented in the metamodel as a specialization of the meta-class `Interface`, called `Event` 5.8.

Metrics definition

We will start by discussing the formalization of component interface complexity assessment metrics, presented earlier, in section 5.3.1. The original specification of the **APP** metric relies on the existence of a single interface for the component being measured. In contrast, in UML 2.0 it is possible for a component to provide several interfaces. So, in order to capture the intention of Boxall and Araban's definition of **APP**, we need to use the union of the provided interfaces, as a surrogate for the "unique interface" used in the original **APP** metric definition.

First, we formalize a few auxiliary functions, for collecting and counting interfaces (`ProvidedInterfaces` and `ProvidedInterfacesCount`, respectively) and operations (`ProvidedOperations` and `ProvidedOperationsCount`, respectively). Note that in this metamodel, events are represented as a stereotyped interface (i.e., as instances of the meta-class `Event`). As such, we need to filter out the instances of `Event` so that we only keep the remaining interfaces, as expected in the metrics we are about to formalize. Then, we define `NA` (representing n_a) and `NP` (representing n_p) as the numerator and denominator of the `APP` fraction, so that the formalized definition has a fairly straightforward mapping to the definition of **APP** presented in equation 5.1.

Listing 5.3: APP metric in UML 2.0.

```
context Component
  ProvidedInterfaces(): Set(Interface) = self.ownedPort.provided->select(i |
    not i.oclIsKindOf(Event))->asSet()
  ProvidedInterfacesCount(): Integer = self.ProvidedInterfaces()->size()

  ProvidedOperations(): Set(Operation) =
    self.ProvidedInterfaces().ProvidedOperations()->asSet()
  ProvidedOperationsCount(): Integer = self.ProvidedOperations()->size()

  NA(): Integer = self.ProvidedOperations().NA()->sum()
  NP(): Integer = self.ProvidedOperationsCount()
  APP(): Real = self.NA()/self.NP()
```

The `ProvidedOperations` function defined in the `Component` metaclass (listing 5.3) uses the `ProvidedOperations` function, defined in the `Interface` metaclass (listing 5.4).

Listing 5.4: Interface provided operations in UML 2.0.

```
context Interface
  ProvidedOperations(): Set(Operation) = self.ownedOperation
  ProvidedOperationsCount(): Integer = ProvidedOperations()->size()
```

The `NA` function definition for the `Component` metaclass (listing 5.3) relies on the counting attributes with the same identifier, defined in the scope of the `Operation` metaclass (listing 5.4).

Listing 5.5: Parameters of operations in UML 2.0.

```
context Operation
  Parameters(): Set(Parameter) = self.ownedParameter
  NA(): Integer = self.Parameters()->size()
```

In order to formalize the definition of the **DAC**, **DAR**, and **ARS** metrics, we start by defining an auxiliary function in the scope of the `Parameter` metaclass that receives a collection of operations and returns `true` if any of those operations contains a parameter with the same name and type of the parameter under scrutiny (listing 5.6).

Listing 5.6: Detecting similar parameters in a set, in UML 2.0.

```
context Parameter
  ExistsNameType(s: Set(Parameter)): Boolean =
    s.exists((self.name=name) and (self.type.name=type.name))
```

We can now formalize DAC, DAR, and ARS in the Component metaclass. Again, we will also define a few auxiliary functions so that the final definition of each of these metrics is more easily traceable to those provided in equations 5.2 through 5.4. The DistinctArguments function returns the set of parameters used in the operations, defined in such a way that there are no repetitions of parameters with the same name and type. This function makes the definition of the DAC function trivial. The ACount function returns the number of operations provided by the component which include a parameter with the same name and type of its argument. It corresponds to a in equation 5.4. Sum_A builds on this definition to represent the sum of the squared a terms, in the same equation, thus facilitating the mapping between the ARS function and the definition in equation 5.4.

Listing 5.7: DAC, DAR and ARS, in UML 2.0.

```
context Component
  DistinctArguments(): Set(Parameter) =
    self.ProvidedOperations().Parameters()->iterate(
      p: Parameter; noDups: Set(Parameter) = oclEmpty(Set(Parameter)) |
      if (not p.ExistsNameType(noDups))
      then noDups->including(p)
      else noDups
      endif)

  DAC(): Integer = self.DistinctArguments()->size()

  DAR(): Real = self.DAC()/self.NA()

  ACount(a: Parameter): Integer = self.ProvidedOperations()->
    select(o | a.ExistsNameType(o.Parameters()))->size()
  Sum_A(): Integer = self.DistinctArguments()->collect(p |
    ACount(p)*ACount(p))->sum()

  ARS(): Real = self.Sum_A()/self.NA()
```

The **CPD** metric can be defined using several alternative constituents, for which we aim to measure the density. Although Narasimhan and Hendradjaya defined this metric for the whole component assembly, we will start by providing a definition for a single component, which we will then reuse when computing **CPD** for the whole component assembly. As discussed in section 5.3.1, we will use the operations made available through the component's provided interfaces as the example of packageable element, in our definition of the CPD function. This makes the definition quite simple, as we can reuse the function ProvidedOperationsCount.

Listing 5.8: CPD in UML 2.0.

```
context Component
  CPD(): Integer = self.ProvidedOperationsCount()
```

The **CPD** metric is also targeted for the whole component assembly, so that we can detect deviations to typical component packing density. As there is no model element representing the model itself, we have to use OCL's `allInstances` function to gain access to the collection of all components, and then compute the average CPD for the whole component assembly. Therefore, unlike in previous specifications, we omit a context expression in listing 5.9. Note that, in OCL, we have to define all the expressions within a context. By omitting the context, we are just emphasizing that, in this particular case, any choice of context would be arbitrary.

Listing 5.9: CPD in UML 2.0.

```
CPD(): Real = Component.allInstances.CPD()->sum()/
  Component.allInstances->size()
```

The **EFI** and **EFO** metrics have a fairly straightforward definition in OCL. The `EFI` function returns the number of events originated by the component. In practice, this corresponds to the provided interfaces which are stereotyped as an event, in the metamodel presented in figure 5.8. Conversely, the `EFO` function returns the number of events that are consumed by the component.

Listing 5.10: EFI and EFO, in UML 2.0.

```
context Component
  ProducedEvents(): Set(Event) = self.ownedPort.provided->iterate(
    e: Interface; result: Set(Event)=oclEmpty(Set(Event)) |
      if (e.oclIsKindOf(Event))
  then result->including(e.oclAsType(Event))
  else result
  endif)

  ConsumedEvents(): Set(Event) = self.ownedPort.required->iterate(
    e: Interface; result: Set(Event)=oclEmpty(Set(Event)) |
      if (e.oclIsKindOf(Event))
        then result->including(e.oclAsType(Event))
        else result
        endif)

  EFI(): Integer = self.ProducedEvents()->size()
  EFO(): Integer = self.ConsumedEvents()->size()
```

This concludes the formalization of the component metrics that support the assessment of components, independently of the component assembly in which the component is integrated.

In order to define component service utilization metrics (discussed in section 5.3.2), we will follow a similar strategy, by defining auxiliary OCL functions that will make

the final metrics definitions as close as possible to their original specification, to allow for an easier mapping between both. As noted in the respective comments section, we map the concept of service to that of interface. In other words, each interface provided, or required, by a component corresponds to a service, in this formalization.

We start by addressing the **PSU** metric. We can define the set of provided services as the set of interfaces provided by the component (`ProvidedServices` function).

The `PTotal` function matches P_{total} in equation 5.6. Defining `PActual` is slightly more complex, as we need to filter out from the available services those which are not being used by any of the components within the assembly. As the associations between components and interfaces are unidirectional (from components to interfaces), we need to query all instances of components, to select which of those components are using a given interface.

A similar approach can be followed in the definition of the **RSU** metric, using the `RequiredServices` function, along with `RTotal` and `RActual`, which stand for R_{total} and R_{actual}, in the original metric's definition provided in equation 5.7. Listing 5.11 condenses the definitions required for computing both `PSU` and `RSU`.

Since we stipulated that a service corresponds to an interface, for the purposes of our formalization of this metric, we could use the `ProvidedInterfaces` (as defined in listing 5.3) and the `RequiredInterfaces` (defined in listing 5.11) functions directly in the definitions of `PSU` and `RSU`. However, as this is just one of the possible mappings of the concept of service in the context of software components, we prefer to create two extra functions, `ProvidedServices` and `RequiredServices`, so that the metrics definition could be easily adjusted to other mappings, by changing only these two functions. In listing 5.11 we start by defining the `RequiredInterfaces` similarly to our earlier definition for `ProvidedInterfaces`. Then, we define the remaining functions used in the specification of `PSU` and `RSU`.

Listing 5.11: PSU and RSU, in UML 2.0.

```
context Component
  RequiredInterfaces(): Set(Interface) = self.ownedPort.required->select(i|
    not i.oclIsKindOf(Event))->asSet()

  ProvidedServices(): Set(Interface) = self.ProvidedInterfaces()
  RequiredServices(): Set(Interface) = self.RequiredInterfaces()

  PActual(): Integer = self.ProvidedServices()->select(s |
    Component.allInstances.RequiredServices()->includes(s))->size()
  PTotal(): Integer = self.ProvidedServices()->size()

  RActual(): Integer = self.RequiredServices()->select(s |
    Component.allInstances.ProvidedServices()->includes(s))->size()
  RTotal(): Integer = self.RequiredServices()->size()

  PSU(): Real = self.PActual()/self.PTotal()
```

```
  RSU(): Real = self.RActual()/self.RTotal()
```

We can now define, for the whole component assembly, both `CPSU` and `CRSU` (listing 5.12). These functions correspond to equations 5.8 and 5.9, respectively. In both definitions, all the component metaclass instances in the model are assumed to be part of the same component assembly. The UML metamodel does not include a model element representing "a model" (in this case, a component assembly) that would contain all the model elements, so this is a fair assumption.

Listing 5.12: CPSU and CRSU in UML 2.0.

```
CPSU(): Real = Component.allInstances.PActual()->sum() /
  Component.allInstances.PTotal()->sum()

CRSU(): Real = Component.allInstances.RActual()->sum() /
    Component.allInstances.RTotal()->sum()
```

Finally, we will discuss the formalization of component interaction density metrics, presented in section 5.3.2. As components interact among themselves through their interfaces and the production, or consumption of events, the definitions are fairly similar to those of service utilization metrics.

We start by defining what incoming and outgoing interactions are, both potential and instantiated in the model, and then build on those functions for defining the interaction density metrics `IIDC`, `OIDC`, and `IDC`, in listing 5.13.

Listing 5.13: Component interaction density metrics in UML 2.0.

```
context Component
  AvailableConsumedEvents(): Set(Event) = self.ConsumedEvents()->select(e |
    Component.allInstances.ProducedEvents()->includes(e))
  AvailableConsumedEventsCount(): Integer =
    self.AvailableConsumedEvents()->size()

  UsedProducedEvents(): Set(Event) = self.ProducedEvents()->select(e |
    Component.allInstances.ConsumedEvents()->includes(e))
  UsedProducedEventsCount(): Integer = self.UsedProducedEvents()->size()

  IncomingInteractions(): Set(Interface) =
    self.ProvidedInterfaces()->union(self.ConsumedEvents())->asSet()
  IncomingInteractionsCount(): Integer = self.IncomingInteractions()->size()

  OutgoingInteractions(): Set(Interface) =
    self.RequiredInterfaces()->union(self.ProducedEvents())->asSet()
  OutgoingInteractionsCount(): Integer = self.OutgoingInteractions()->size()

  IIN(): Integer = self.AvailableConsumedEventsCount() + self.RActual()
  IMaxIn(): Integer = self.ConsumedEventsCount() + self.RTotal()

  IIDC(): Integer = self.IIN() / self.IMaxIn()
```

```
  IOUT(): Integer = self.UsedProducedEventsCount() + self.PActual()
  IMaxOut(): Integer = self.ProducedEventsCount() + self.PTotal()

  OIDC(): Real = self.IOUT() / self.IMaxOut()

  I(): Integer = self.IIN() + self.IOUT()
  IMax(): Integer = self.IMaxIn() + self.IMaxOUT()

  IDC(): Real = self.I() / self.IMax()
```

We can formalize the **AIDC** metric using a similar strategy to the one followed for **CPSU** and **CRSU**. Listing 5.14 presents the definition for the `AIDC` function.

Listing 5.14: AIDC in UML 2.0.

```
AIDC(): Real =
  Component.allInstances.IDC()->sum() / Component.allInstances->size()
```

Throughout these metrics definitions, it was often the case (e.g. in the `AIDC` metric definition) that the used formula contains a ratio. In these situations, we can choose one of two alternatives:

- We can keep the definition as it is, and leave it to the OCL analyzer to deal with the situations where the denominator equals 0, in its standard way. For example, the OCL tool used in the preparation of dissertation (USE[2]) returns `undefined` in these situations. `undefined` acts as the neutral element in further functions where that result is required. For instance, when computing the sum of a number of variables, if one of those variables is `undefined`, the tool substitutes the value `undefined` by 0.

- We can define pre-conditions for the functions, to prevent computations to be performed when they do not make sense.

The definitions presented so far followed the first option. We will now explore how they can be complemented to conform to the second alternative. Consider, for instance, the `IDC` function definition. If we add a pre-condition, as in listing 5.15, we are explicitly preventing the `IDC()` function to be computed when `self.IMax() > 0`. Note also that the specification of context includes the function name qualified by the owner metaclass.

Listing 5.15: Adding metrics pre-conditions in UML 2.0.

```
context Component::IDC(): Real
  pre: self.IMax() > 0
  post: result = self.I() / self.IMax()
```

[2] http://www.db.informatik.uni-bremen.de/projects/USE/

In listing 5.16, we add the pre-conditions to each of the functions, but omit the post-conditions, to avoid their repetition from the previous listings. All the presented pre-conditions relate to avoiding divisions by 0, but the same principle can be applied to other restrictions we might want to impose concerning metrics definitions.

Listing 5.16: Metrics pre-conditions in UML 2.0.

```
context Component::APP(): Real
  pre: self.NP() > 0

context Component::DAR(): Real
  pre: self.NA() > 0

context Component::ARS(): Real
  pre: self.NA() > 0

context Component::PSU(): Real
  pre: self.PTotal() > 0

context Component::RSU(): Real
  pre: self.RTotal() > 0

context Component::IIDC(): Real
  pre: self.IMaxIn() > 0

context Component::OIDC(): Real
  pre: self.IMaxOut() > 0

context Component::IDC(): Real
  pre: self.IMax() > 0

context ::CPSU(): Real
  pre: Component.allInstances.PTotal()->sum() > 0

context ::CRSU(): Real
  pre: Component.allInstances.RTotal()->sum() > 0

context ::AIDC(): Real
  pre: Component.allInstances->size() > 0
```

5.4.2 CORBA Component Metamodel

Context specification

The Corba Component Metamodel (CCM) [OMG 02a, Wang 01, Estublier 02] is the OMG standard for the specification of software components. As such, it is independent from a specific vendor, both in what concerns the component's programming languages and platforms. The CCM specification includes a Meta Object

Facility-compliant metamodel [OMG 02b][3], where the CCM modeling elements are precisely defined. The metamodel includes three packages (figure 5.9). The **BaseIDL** package contains the modeling elements concerning the **CORBA Interface Description Language** (**IDL**). **BaseIDL** is extended by **ComponentIDL**, to add the component specific constructs. Finally, **ComponentIDL** is extended by the **C**omponent **I**mplementation **F**ramework (**CIF**) package, which includes the definitions relating to the component lifecycle.

Figure 5.9: CCM packages (adapted from [OMG 02a])

The CCM metamodel has some characteristics that are relevant for the formalization presented in this paper, namely the synchronous and asynchronous interaction mechanisms among components, presented in figure 5.10.

The synchronous interaction mechanism is specified as follows: each CORBA component may have several provided (facets) and required (receptacles) interfaces, represented in the metamodel through the `ProvidesDef` and `UsesDef` metaclasses, respectively. Both use the `InterfaceDef` metaclass to support the specification of the interfaces being provided or required.

The asynchronous mechanism uses events, represented in the metamodel as `EventDef`, to support interactions among components. Components may specify the events they produce and consume through ports, specified in the metamodel as `EventPortDef`. There are two alternatives that can be used while producing an event: the event may be emitted (`EmitsDef`) for a single consumer, or published (`PublishesDef`) for a set of potential consumers. Components may consume events (both emitted and published) by subscribing to the event sources (`ConsumesDef`).

Metamodel Extension

Similarly to the UML 2.0 metamodel, the CCM metamodel includes abstractions for representing CORBA components, but not for the representation of component assemblies. For instance, it is possible to express that component **A** has facets **X** and **Y**, and that component **B** has facet **Z** and receptacle **Y**, but there is no way of expressing that **A**'s facet **Y** is wired to **B**'s receptacle **Y**, with the standard metamodel. This wiring is specified through a **Component Assembly Descriptor** file. A **D**ocument **T**ype **D**efinition (**DTD**) for such files can be found in [Merle 03].

[3]Note that although the referenced CCM 3.0 ([OMG 02a]) uses an older version of MOF, MOF has evolved since then. A more recent version of MOF is available in [OMG 04]. Our discussion in this section refers to the MOF version used in CCM 3.0.

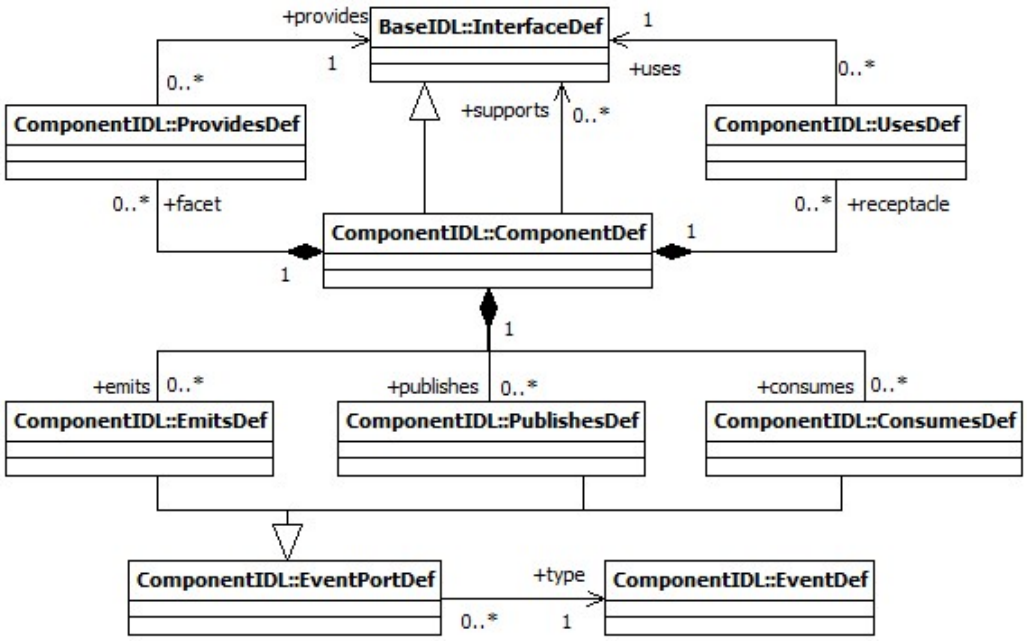


Figure 5.10: Excerpt of the CCM (adapted from [OMG 02a]).

In [Goulão 05a] we extended the CCM metamodel to overcome this shortcoming, by adding a new package called **Component Assembly Metamodel Extension** (**CAME**), as shown in figure 5.11.

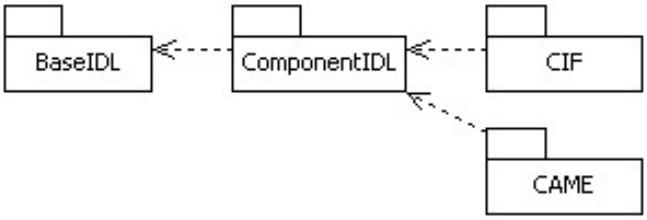


Figure 5.11: Extended CCM model

In figures 5.12 through 5.14, we represent the origin of all meta-classes, except for those defined in the **CAME** package. Figure 5.12 represents the extensions required for wiring components through their provided and used interfaces. These include **ComponentInstanceDef** and **InterfaceConnectorDef** for representing component instances in a component assembly, and a connector between provided and used interfaces ports (**ProvidesPortDef** and **UsesPortDef**).

With these new meta-classes, and the associations among them, we have at our disposal the required abstractions for representing component assemblies, including the ports and connectors used for component instances wiring. Figure 5.13 includes the meta-classes required for wiring an event emitter with an event consumer. The modeling approach is similar to what was described for provided and required interfaces.

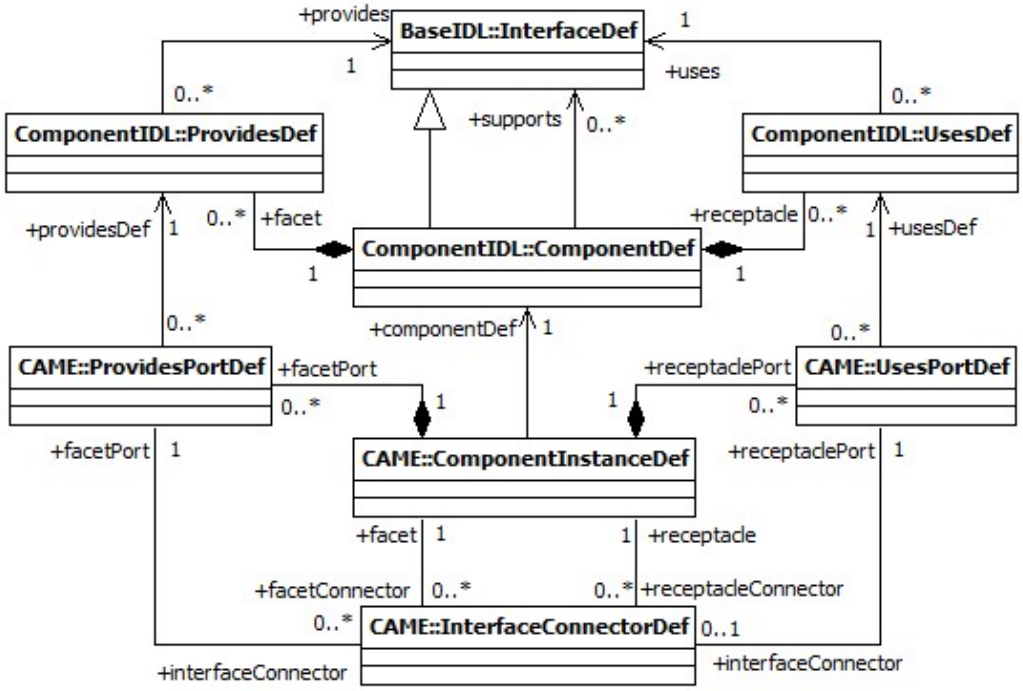


Figure 5.12: Metamodel extensions for component wiring through provided and used interfaces

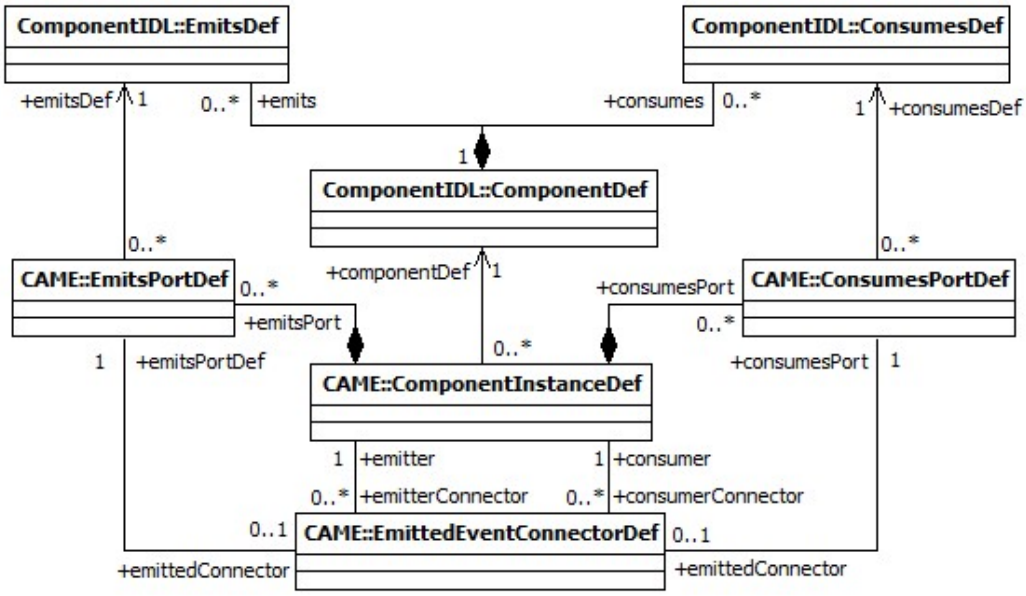


Figure 5.13: Metamodel extensions for component wiring through emitted events

Figure 5.14 includes the meta-classes for representing the wiring between an event broadcaster and the event's consumers.

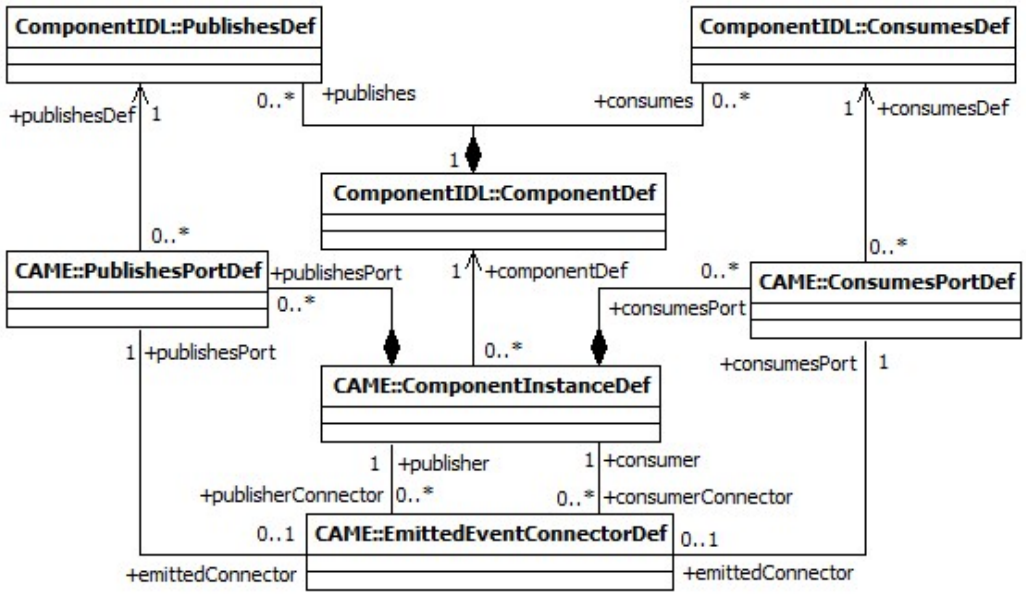


Figure 5.14: Metamodel extensions for component wiring through published events

Finally, we need an abstraction to represent the component assembly. A component assembly may have an arbitrary number of component instances (Figure 5.15).

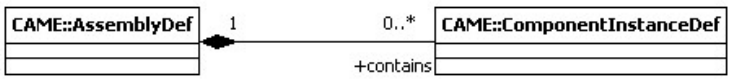


Figure 5.15: The component assembly metaclass

Metrics definition

The definition of the **APP** metric relies on the existence of a single interface for the component. There is no directly equivalent modeling element in the CCM metamodel. The component equivalent interface is broader, as it includes all implicit operations (a set of operations defined by component homes), operations and attributes which are inherited by the component (also through supported interfaces) and attributes defined inside the component. On the other hand, considering just a single interface as the context would lead to a metric different than the one proposed by Boxall and Araban. To be precise in our formalization, we will consider all the procedures from all the provided interfaces of the component, when computing Boxall and Araban's metrics.

The context for Boxall and Araban's metric definition is `ComponentDef`. We start by defining `ProvidedOperations`, the set of operations used in the metrics definition,

and ProvidedOperationsCount, the cardinality of this set. The formalization of the APP metric becomes straightforward, with these auxiliary functions.

Listing 5.17: The APP metric in CCM.

```
context ComponentDef
  ProvidedInterfaces(): Set(InterfaceDef) = self.facet.provides->asSet()
  ProvidedInterfacesCount(): Integer = self.ProvidedInterfaces()->size()

  ProvidedOperations(): Set(OperationDef) =
    self.ProvidedInterfaces().Operations()->flatten()->asSet()
  ProvidedOperationsCount(): Integer = self.ProvidedOperations()->size()

  NA(): Integer = self.ProvidedOperations().NA()->sum()
  NP(): Integer = self.ProvidedOperationsCount()
  APP(): Real = self.NA()/self.NP()
```

As in the UML metrics formalization, we are using a ProvidedOperations function in the InterfaceDef class (listing 5.18).

Listing 5.18: Interface provided operations in CCM.

```
context InterfaceDef
  Operations(): Set(OperationDef) = self.contents->select(o |
    o.oclIsKindOf(OperationDef))->collect(oclAsType(OperationDef))->asSet()
  OperationsCount(): Integer = self.Operations()->size()
```

We can obtain the parameters of each operation (listing 5.19), to support the definition used in the Operations function specification in the context of the OperationDef metaclass (listing 5.18).

Listing 5.19: Parameters of operations in CCM.

```
context OperationDef
  Parameters(): Set(ParameterDef) = self.parameters->asSet()
  NA(): Integer = self.Parameters()->size()
```

In order to facilitate the detection of similar parameters in a component's interface, we will start by defining the ExistsNameType in the context of the ParameterDef metaclass, following, again, the same strategy as in the UML metrics formalization (listing 5.20). This function returns true if a duplicate of the parameter is found in a set of parameters.

Listing 5.20: Detecting similar parameters in a set, in CCM.

```
context ParameterDef
  ExistsNameType(s:Set(ParameterDef)): Boolean =
    s->exists((self.identifier=identifier) and (self.idlType = idlType))
```

In the context of ComponentDef we can now define DistinctArguments to return the list of arguments used in the provided interfaces operation signatures, without duplicates. DAC computes the distinct arguments count and DAR their percentage in the

component interface. Finally, we define two auxiliary functions, `ACount` and `Sum_A`, which compute the count of procedures in which the argument is used, and the sum of the squares of `ACount`. These functions are defined in listing 5.21.

Listing 5.21: Component interface metrics.

```
context ComponentDef
  DistinctArguments(): Set(ParameterDef) =
    self.ProvidedOperations().Parameters()->iterate(
      p: ParameterDef; noDups: Set(ParameterDef) =
        oclEmpty(Set(ParameterDef)) |
          if (not (p.ExistsNameType(noDups)))
          then noDups->including(p)
          else noDups
          endif)

  DAC(): Integer = self.DistinctArguments()->size()

  DAR(): Real = self.DAC()/self.NA()

  ACount(a: ParameterDef): Integer = self.ProvidedOperations()->
    select(o: OperationDef | a.ExistsNameType(o.Parameters()))->size()
  Sum_A (): Integer = self.DistinctArguments()->collect(p|
    ACount(p)*ACount(p))->sum()

  ARS(): Real = self.Sum_A()/self.NA()
```

The rationale for the formalization of the **CPD** metric is the same as the one used in the UML formalization, and so is the option concerning the chosen constituents, to facilitate the comparison between both definitions. We will start by defining `CPD` in the context of a single component, in listing 5.22.

Listing 5.22: The CPD metric in CCM.

```
context ComponentDef
  CPD(): Integer = self.ProvidedOperationsCount()
```

The formalization of `CPD` for the component assembly is performed using a different context than the previous ones. We will use a CCM module (represented by `ModuleDef` in the metamodel), rather than an individual component, as the context for this definition (listing 5.23).

Listing 5.23: The CPD metric in CCM.

```
context ModuleDef
  Components(): Set(ComponentDef) =
    self.contents->select(oclIsKindOf(ComponentDef))->
    collect(oclAsType(ComponentDef))->asSet()

  ComponentsCount(): Integer = self.Components()->size()
```

```
  ConstituentsCount(): Integer =
    self.Components().ProvidedOperationsCount()->sum()

  CPD(): Real = self.ConstituentsCount()/self.ComponentsCount()
```

The CCM metamodel distinguishes between published and emitted events. The former are made available to several event consumers, while the latter are made available to a single listener. For the purposes of the EFI function, this distinction is not relevant. Therefore, when counting the events made available by a component, we will simply add the published and the emitted events. No distinction is made concerning the consumer of the events. Listing 5.24 formalizes these definitions.

Listing 5.24: The EFI and EFO metrics in CCM.

```
context ComponentDef
  Emits(): Set(EmitsDef) = self.emits
  EmitsCount(): Integer = self.Emits()->size()

  Publishes(): Set(PublishesDef) = self.publishes
  PublishesCount(): Integer = self.Publishes()->size()

  Consumes(): Set(ConsumesDef) = self.consumes
  ConsumesCount(): Integer = self.Consumes()->size()

  EFI(): Integer = self.PublishesCount() + self.EmitsCount()
  EFO(): Integer = self.ConsumesCount()
```

Concerning the metrics which depend on the component assembly in which the components are integrated, we will need to use the CCM metamodel extension, discussed in section 5.4.2.

A CCM assembly can be represented as an instance of the extended CCM metamodel. This instance can be seen as a directed graph of meta-objects (nodes) representing the modeling elements used in the assembly, and the appropriate meta-links (edges) among them.

Listing 5.25: PSU and RSU metrics in CCM.

```
context ComponentInstanceDef
  ProvidedServices(): Set(InterfaceDef) =
    self.facetConnector.facetPort.providesDef.provides->asSet()
  RequiredServices(): Set(InterfaceDef) =
    self.receptacleConnector.receptaclePort.usesDef.uses->asSet()

  PActual(): Integer = self.ProvidedServices()->size()->select(s |
    ComponentInstanceDef.allInstances.RequiredServices()->includes(s))->
    size()
  PTotal(): Integer = self.ProvidedInterfaces()->size()
  RActual(): Integer = self.RequiredServices()->size()
    ComponentInstanceDef.allInstances.ProvidedServices()->includes(s))->
```

```
    size()
  RTotal(): Integer = self.RequiredInterfaces()->size()

  PSU(): Real = self.PActual()/self.PTotal()

  RSU(): Real = self.RActual()/self.RTotal()
```

The formalization for the metrics on the component assembly is defined in the context of the `Assembly` metaclass as in listing 5.26.

Listing 5.26: Assembly metrics in CCM.

```
context Assembly
  Components(): Set(ComponentInstanceDef) = self.contents->select(c |
    c.oclIsKindOf(ComponentInstanceDef))->asSet()

  CPSU(): Real = self.Components().PActual()->sum() /
    self.Components().PTotal()->sum()

  CRSU(): Real = self.Components().RActual()->sum() /
    self.Components.RTotal()
```

The next listing contains the formalization of the IDC, IIDC, and OIDC metrics, all in the context of the metaclass `ComponentInstanceDef`.

Listing 5.27: The IDC, IIDC, and OIDC metrics in CCM.

```
context ComponentInstanceDef
  EmittedEventsConsumed(): Set(EmitsDef) =
    self.emitterConnector.emitsPort.emitsDef->asSet()

  EmittedEventsConsumedCount(): Integer =
    self.EmittedEventsConsumed()->size()

  EmittedEvents(): Set (EmitsDef)= self.componentDef.emits

  EmittedEventsCount(): Integer = self.EmittedEvents()->size()

  PublishedEventsConsumed(): Set (PublishesDef)=
    self.publisherConnector.publishesPort.publishesDef->asSet()

  PublishedEventsConsumedCount(): Integer =
    self.PublishedEventsConsumed()->size()

  PublishedEvents(): Set (PublishesDef)= self.componentDef.publishes

  PublishedEventsCount(): Integer = self.PublishedEvents()->size()

  ProducedEventsCount(): Integer = self.EmittedEventsCount() +
    self.PublishedEventsCount()
```

```
  UsedProducedEventsCount(): Integer =
    self.PublishedEventsConsumedCount() + self.EmittedEventsConsumedCount()

  AvailableConsumedEvents(): Set(ConsumesDef) =
    self.emitConsumerConnector.consumesPort.consumesDef->union(
    self.publishConsumerConnector.consumesPort.consumesDef)->asSet()

  AvailableConsumedEventsCount(): Integer =
    self.AvailableConsumedEvents()->size()

  ConsumedEvents(): Set(ConsumesDef) = self.componentDef.consumes

  ConsumedEventsCount(): Integer = self.ConsumedEvents()->size()

  IIn(): Integer = self.AvailableConsumedEventsCount() + self.RActual()
  IMaxIn(): Integer = self.ConsumedEventsCount() + self.RTotal()

  IIDC(): Real = self.IIn()/self.IMaxIn()

  IOut(): Integer = self.PublishedEventsConsumedCount() +
    self.EmittedEventsConsumedCount() + self.PActual()
  IMaxOut(): Integer = self.ProducedEventsCount() + self.PTotal()

  OIDC(): Real = self.IOut()/self.IMaxOut()
  I(): Integer = self.AvailableConsumedEventsCount()
    + self.PublishedEventsConsumedCount()
    + self.EmittedEventsConsumedCount()
    + self.PActual() + self.RActual()

  IMax(): Integer = self.ConsumedEventsCount() + self.PublishedEventsCount()
    + self.EmittedEventsCount() + self.PTotal() + self.RTotal()

  IDC(): Real = self.I()/self.IMax()
```

Finally, the AIDC metric can be formalized as represented in listing 5.28.

Listing 5.28: The AIDC metric in CCM.

```
context Assembly
  AIDC(): Real = self.Components().IDC()->sum()/self.Components()->size()
```

5.5 Comments on the metrics' definitions

5.5.1 Uncovering shortcomings in the original metrics definitions

We deliberately kept the original formulas to demonstrate the variability of notations that are commonly used in metrics definitions. A "side-effect" of these informal approaches is that the metrics formula is often not specified in extreme cases. The origi-

nal definitions are not clear, regarding how the situations marked as **NA** in tables 5.1 and 5.2 are treated. A plausible explanation is that these situations may be considered as either (i) having an "obvious" solution that can be left out in the definition and postponed as an "implementation detail", or (ii) completely forgotten. The problem with (i) is that the mapping of "obvious" to a concrete solution may vary according to the background of the practitioner implementing the metrics collection tool, thus leading to inconsistent metrics collection performed by different tools. The obliviousness resulting from (ii) leads to a similar situation.

The ODM approach mitigates these problems not only due to its increased formality, but also because the metrics are executable. As such, they can be tested as soon as they are specified. It is more natural for a practitioner to be concerned about defining how to handle those extreme cases, as neglecting to do so is likely to expose the limitations of the definition earlier (when compared to defining something and not being able to automatically test that definition).

5.5.2 Reusing formalizations

By using auxiliary functions for performing the most basic counts, the formalized definitions of each of these metrics becomes almost independent from the underlying metamodel, which is a benefit in terms of the portability of the definitions for other metamodels. For instance, although the counting rules for model elements such as provided and required interfaces differ from UML to CCM, due to metamodel differences, functions built upon those auxiliary counting functions are very similar, except for the metaclass they are defined upon.

To make the metrics completely independent from the specific metamodel, we would have to use an independent metamodel that could be mapped to the more specific ones. In the realm of CBD, this would mean having a *component model independent metamodel*. To the best of our knowledge, there is no currently widely accepted metamodel that can be used. In the realm of ADL-based component models, a metamodel for Acme [Goulão 03] could be used. UML 2.0, extended with profiles, is another possible candidate.

As we have seen, depending on what one is trying to measure, a metamodel may, or may not, be suitable. Rather than trying to have an all-purpose metamodel, an alternative is to define a smaller metamodel, focused in modeling a specific concern. One such example is the PIMETA [Bryton 07, Bryton 08], a paradigm independent metamodel designed to represent modularity information required to support modularity assessment metrics. The price to pay for this generality is that it may be the case that the information loss resulting from the mapping between the specific metamodels and the generic one is a threat to the validity of comparisons performed with different models.

5.5.3 Uncovering hidden relationships between metrics sets

While formalizing metrics definitions, some of the formalized metrics sets turned out to be a lot closer than what one may infer from their presented rationales. This is particularly noticeable with service utilization metrics, when compared to interaction density metrics. The service utilization metrics concern the level of usage (when compared to the maximum possible level) of interaction mechanisms among components, while the interaction density among components is concerned with the number of interactions. Both sets are based on essentially the same information, although each of them has its own quality concern: architectural mismatch, in the case of service utilization metrics, or interaction complexity, in the case of interaction density metrics. This suggests a potential (even if partial) overlap between both concerns.

More generally, the fact that we have to express a metric definition in terms of the metamodel induces a uniformity in terms of the basic counting elements that is often missing in more informal metrics definitions.

5.5.4 Metrics definition patterns

Several of the formalized metrics for the whole component assembly (CPSU, CRSU, and AIDC) are defined as ratios where the numerator corresponds to the effective usage of a given mechanism, while the denominator has the maximum possible utilization of the mechanism within the component assembly. This indicates a concern from the metrics proponents to make them dimensionless. This prevents the metrics values from being correlated to the size of the assembly, or the number of times a particular mechanism is used, and conforms to Abreu's criteria on software metrics, according to which, non-size metrics should be size independent [Abreu 94b].

In contrast, other metrics are defined as basic counts, or average values of those counts. They are typically related to complexity metrics, and often correlated with size. Size metrics have no practical value, *per se*. They can be made useful, however, either by correlating their values to another property of the element under scrutiny (e.g. the number of defects found in it), or by comparing that value with typical values for that metric in other components and further investigating those elements which present atypical metric values.

5.5.5 Quality framework

Without a clear notion of the quality attributes we wish to assess and the criteria we will use to interpret the metrics values, it is not possible to analyze the results. Although the authors of the proposed metrics provide a rationale for them, the lack of a well-defined quality framework is noticeable.

When analyzing the values presented in tables 5.1 through 5.3, based on the rationale presented during their formalization, one can only make relative judgments on

their values. For instance, from the point of view of these metrics, the understandability of the component **Clock** is lower than that of **ParkingCamera** in what concerns **APP** and **DAC**. However, this higher complexity is mitigated by the tendency for repeating more arguments (**ARS**), in the **ParkingCamera** component. With respect to the CPD metric, **Clock** is, again, regarded as a more complex component than **ParkingCamera**.

It is impossible to make judgments concerning the absolute complexity of these components (e.g. is a component with a **CPD** excessively complex?), in the absence of a quality framework.

5.5.6 Metrics definition context

The lack of an adequate metamodel in the original metrics definitions justifies our need to include several comments on the assumptions made before formalizing each metric (see the **Comments** section of all metrics descriptions). A metamodel clarifies the used concepts and their interrelationships, providing a backbone upon which we can formalize the metrics definitions with OCL. The combination of the metamodel with the OCL expressions removes the subjectivity from the metrics definitions. The metamodel is also useful for the automation of metrics collection.

5.5.7 Specification formalism

We deliberately used the original formalisms in metrics definitions (see the **Original specification** section of all metrics formalization) to illustrate their diversity. For instance, the concept of collection size is conveyed with three different notations in equations 5.1 through 5.13: a plain identifier (e.g. n_a), an identifier between a pair of '|' characters (e.g. $|A|$), and the # notation (e.g. #<constituents>). Equation 5.4 uses simultaneously two of these notations. This may lead to misinterpretations of the formulae.

Ambiguity resulting from the usage of natural language is also a problem. Suppose that rather than counting provided operations as constituents for the **CPD** metric, we would like to count provided interfaces. It is possible for different components to provide the same interface. In that case, should we count it once, or several times? If we use the informal version of the definition, we might just write $constituent\_type = provided\ interface$ and be left with an ambiguous definition.

Now, consider the two alternative `ConstituentsCount` function definitions in listing 5.29:

Listing 5.29: The ConstituentsCount function in CCM.

```
context ModuleDef
  -- Constituents as Interfaces with duplicates
  ConstituentsCount(): Integer =
    self.Components()->collect(ProvidesCount())->sum()
```

```
-- Constituents as interfaces without duplicates
ConstituentsCount(): Integer =
  self.Components()->collect(ProvidesNoDupsCount())->sum()
```

From the formal definition, it is clear that what we mean is *several times* in the first version and *once* on the second one, thus removing the ambiguity. A similar argument can be made for several of the metrics formalized in this chapter.

5.5.8 Computational support

Most of the computational support required for collecting metrics defined in OCL is either publicly available, or can be built with relatively low effort. The core of the computational support consists of an OCL-enabled UML tool (e.g. the USE tool), with the ability to load a metamodel (as a class model) and create instances of those models (e.g., using object diagrams). In the first formalization, we need to load the UML 2.0 metamodel and populate it with the appropriate instances, representing the UML components. In the second formalization, we load the CCM metamodel and then populate it to represent the CORBA components.

The instantiation of the metamodel can be done manually in a UML tool by creating a meta-object diagram. However, it is more practical and scaleable to develop a component that generates the instantiation from the original component's specifications. In our case, as we are are using the USE tool, the component generates a USE script which, in turn, creates the appropriate instantiation of the metamodel.

5.5.9 Flexibility

By specifying the metrics definitions with OCL we have completely removed the code tangling between the metrics definitions and the tool computing the metrics. The metrics definitions are loaded in the UML tool just as any other OCL expression. Tailoring the metrics set to one's specific needs is, then, a matter of writing new OCL functions, similar to those presented in this chapter.

5.5.10 Validation

To the best of our knowledge, none of the metrics presented in this chapter has undergone a thorough validation, so far. Due to the challenges presented in sections 5.5.5 through 5.5.9, it should become clear that the ideal conditions for independent scrutiny of these metrics were not present in their original definitions. Several plausible interpretations could be provided for each definition, and no tool support was available to collect them. These conditions hampered experimental replicability. The ODM approach facilitates independent validation efforts, thus supporting the comparability of results.

5.6 On the complexity of metamodels

As noted by Ma *et al.* [Ma 04], the UML metamodel has been growing, from each version to the next, and has 260 meta-classes in version 2.0. Table 5.4 presents, for several versions of the UML metamodel, a set of architectural complexity metrics, borrowed from Ma *et al.*'s paper. These include:

- **DSC** - Number of meta-classes
- **MNL** - Maximum number of the level of inheritance
- **NMI** - Number of multiple inheritance meta-classes
- **ADI** - Average depth of the meta-classes inheritance structure
- **AWI** - Average width of the meta-classes inheritance structure

Metric	UML 1.1	UML 1.3	UML 1.4	UML 1.5	UML 2.0
DSC	120	133	192	194	260
MNL	6	6	7	7	9
NMI	5	7	7	7	18
ADI	2.46	2.45	2.92	2.93	3.87
AWI	0.77	0.81	0.89	0.89	0.95

Table 5.4: Evolution of the UML 2.0 metamodel.

The complexity metrics presented in table 5.4 show an increase not only in the sheer number of meta-classes, but also on the complexity of their hierarchy. This complexity growth has a positive and a negative impact in ODM. On the one hand, UML 2.0 has now a more detailed metamodel, which includes several new features (e.g. the representation of software component architectures was significantly improved). This improves the metamodel's coverage with respect to potential metrics definitions requirements. On the other hand, the added complexity makes the metamodel more difficult to master, with a specification of over 900 pages [OMG 05b, OMG 06b].

Now, consider the metamodel extract in figure 5.8. This extract of the metamodel only represents 21 metaclasses and (including the 3 enumerations). This represents about 8% of the classes in the metamodel, which correspond to the classes we had to explore to build this metrics set formalization. This raises a methodical question, with respect to the usability of ODM. Should we use a complex metamodel, such as the UML 2.0, or is it better to use a small, focused, metamodel?

To solve this dilemma one should consider at least the following factors:

- Is there tool support for the candidate metamodels? If not, what is the effort required to build such tool support?
- What is the effort required for understanding the metamodel in order to be able to adequately define the metrics?

- Can we focus on a relatively small subset of the candidate metamodel, to decrease the effort required for mastering it?

As we have seen, only a limited portion of the UML 2.0 metamodel (or a fairly similar number of metaclasses, when formalizing metrics upon the CCM) was required for our metrics definition. This somewhat mitigates the learning curve of the metamodel required for the metrics definition. The tool support for the UML metamodel and OCL is available, although it is often the case that tools only support a simplified version of the UML metamodel (e.g. Together Architect [4] and USE , and may not necessarily allow using OCL to query its own metamodel.

An alternative is to create smaller metamodels, and develop tools to populate those metamodels with meta-objects, obtained from the components. This was the alternative taken in this dissertation.

5.7 Conclusions

In this chapter, we explored the expressiveness of the UML 2.0 and CCM 3.0 metamodels with respect to the formal definition of metrics for CBD, using OCL expressions. We formally defined several metrics found in the literature, so that the resulting set covers most of the composition mechanisms used in current component models.

The metrics formalized here were proposed by several authors and include (i) metrics applicable to components in isolation, (ii) metrics applicable to components in the context of specific component assemblies, and (iii) metrics applicable to the component assemblies themselves.

While metrics of the first kind may somehow help component integrators in their selection process, the current components marketplace has not yet achieved the point where quasi-equivalent parts are available from multi-vendor parties as it is common in other engineering fields. Therefore, we believe that metrics for components within assemblies and for component assemblies will be much more useful in the short term, by facilitating the evaluation of the resulting software architectures. They can help in the evaluation and comparison of alternative design approaches, on the identification of cost effective improvements and on long term financial planning (total cost of ownership), allowing the computation of estimates on deployment and evolution costs.

We discussed our technique with respect to the mitigation of recurrent problems with metrics definitions (lack of a quality framework, lack of an ontology, inadequate specification formalism, computational support, flexibility, and insufficient validation).

Having a formal and executable definition of metrics for component assemblies is an enabling precondition to allow for independent scrutiny of such metrics, when

[4] http://www.borland.com/us/products/together/index.html

combined with an adequate quality framework. While the provided metrics formalization is in itself a contribution to such an independent scrutiny, the formalization technique is amenable to the definition of new metrics, not only for UML 2.0 and CCM assemblies, but also for other component models.

Chapter 6

Process assessment in CBD

Contents

Background: *Although developing reusable components has its own specific concerns when compared to other approaches to software development, some processes, such as code inspections, are generic to software development. Understanding the drivers for inspection success can help improving code inspection activities.*

Objective: *To assess the influence of practitioners' expertise in code inspection of software components.*

Method: *Subjects expertise is determined based on their independently assessed academic record. Inspection outcome is represented by the diversity of defects found, using two alternative metrics. Correlation and tests for detecting significant differences in several populations are used to verify our hypotheses.*

Results: *We found statistically significant relationships among expertise and inspection outcomes. The usage of peer reviewers with a higher expertise than that of the inspected artifact's developers lead to a higher diversity of defects found.*

Limitations: *In an industrial context, an alternative expertise assessment technique should be used, such as the professional experience of the inspection participants.*

Conclusion: *The effect of expertise is observable in the inspection outcome.*

6.1 Motivation

The competence of the members of a software development team is often regarded as a critical success factor for a software project. This observation is not specific to software development, of course. It crosscuts our view on success factors in the society, in general. We expect skilled developers to produce better software than less skilled ones. From requirements definition to the maintenance process of software, the skills of practitioners are a fundamental asset to be considered when planning human resources assignment to projects.

If a typical distribution of competence of the practitioners in an organization is assumed, the distribution of tasks among practitioners will imply that both the most skilled and the less skilled practitioners will have their roles to play in the development process. The challenge, then, is to leverage the expected positive effect of the most skilled practitioners, while limiting the possibly negative effects of less skilled ones.

In this chapter, we will focus on code inspections, a software process activity that is expected to have a strong impact on the final product's quality. We will assess the impact of practitioner's skills on the success of this activity. A **code inspection** is a peer review of source code intended to detect defects before the testing phase begins, thus improving overall code quality. There are a number of code inspection processes being used in industry. Fagan inspections [Fagan 76, Fagan 86] are considered seminal in this area. Fagan defined inspections as a *"**formal**, **efficient**, and **economical** method for finding errors in design and code"*.

Several factors drive code inspection success. Identifying the most influential factors is key to improving inspection effectiveness. Ultimately, the Software Engineering community needs to identify opportunities to improve the return on investment in inspection activities. This motivated the proposal of alternative inspection processes, most of which are evolutions of Fagan inspections. These alternative code inspection techniques try to lower the costs involved in code inspections without sacrificing their benefits. Examples include conducting inspections off-line, thus skipping the inspection meeting [Parnas 87], or performing phased inspections, where the inspectors focus on a specific class of defects [Knight 93], although the latter technique has been criticized for being more costly than conventional inspections [Porter 97b].

In this chapter we assess the impact of practitioners' skills in the context of code inspections performed within a component-based software development process. The code inspections are performed on components developed in-house, thus making their source code available to the code inspections participants. This is a typical scenario in an organization engaged in the development of reusable components that includes code inspections in its development process. Understanding how different combinations of expertise levels influence the outcome of code inspections can help improving the process of selecting effective code inspection teams.

6.1.1 Problem statement

We are concerned with the impact of practitioners' expertise in the outcome of code inspections performed during the development of software components. We are seeking evidence on possible causal relationships between the expertise of practitioners involved in the code inspections and the diversity of defects reported during those inspections, as outlined in figure 6.1. In this scenario, all inspections are carried out by a review team (**RT**) which includes the development team (**DT**) and a peer team (**PT**). This scenario is in line with industry inspections, where the review team includes both the authors of the artifact under scrutiny and other reviewers who act as external auditors for the sake of identifying problems in the artifact. Both the authors and their peers can be (and often are) members of the same organization. The peers are typically other developers who are not involved in the development of the component under scrutiny.

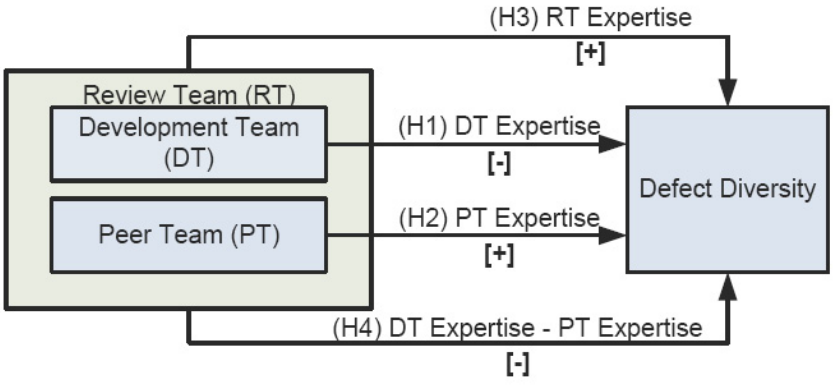


Figure 6.1: Expected expertise impact on the review process

The team dynamics of inspections are likely to play an important role on the inspections' outcome. We consider four potential causal relationships:

- **H1** The expertise of the developer team may have a negative effect on the diversity of defects found. The rationale is that expert developers tend to introduce fewer and less varied defects on their code than other developers.

- **H2** Conversely, the expertise of the peer team may have a positive effect on the defects diversity, as expert peer teams are expected to be better at detecting defects.

- **H3** A similar rationale leads to the possible causal effect between the expertise of the review teams, as a whole, and defect diversity.

- **H4** Finally, we consider the difference of expertise between the developer team and the peer team as a negative effect on defect diversity. If the expertise of the

developers is higher than that of their peers, defect diversity is expected to be smaller than when the opposite occurs.

Understanding how the expertise level of reviewers impacts the diversity of problems found in those inspections, may help improving the review team selection process, by providing information concerning which combinations of reviewers are more effective.

6.1.2 Research objectives

Our goal is to
analyze the outcome of software components source code inspections,
for the purpose of their evaluation,
with respect to the impact of practitioner's expertise on defect introduction and detection,
from the point of view of a project manager (in this case, the research team),
in the context of an academic simulation of a component marketplace.

6.1.3 Context

To better understand the synergies between participants of an inspection we devised an experiment to be run, in this instance, in an academic context, using a toy example. With minor adjustments concerning the expertise assessment of participants (the independent assessment of participants expertise would be replaced by an expertise evaluation scheme in place in the organization), the experimental design is completely reusable in a professional context, with real projects.

The experiment was carried out in the context of a Software Engineering course for 4^{th} year students of the Informatics degree of Universidade Nova de Lisboa, in the Spring semester of 2005. The tasks under scrutiny in this experiment were part of the normal activities within the course, where the students performed the construction of software components, from their requirements' definition to their integration in a component-based system made of components developed by several independent teams.

The participants were novices with respect to performing Code inspections, but already had several years of practice with developing software using the Java programming language. Although the software under inspection was part of a toy example in component-based development, this is not, in itself, a threat to the generalizability of the results. Fagan inspections are usually performed on fairly small portions of source code, with a complexity comparable to that of the source code used in the inspections performed in this experiment. In the design planning section (6.3) we will detail these

issues. Their impact on the potential threats to the validity of the experiment is discussed in sub-section 6.6.2.

6.2 Related work

6.2.1 Inspection techniques

In this section, we provide a brief overview of software inspections techniques, which condenses the observations reported in Laitenberger's survey on inspections [Laitenberger 02].

When discussing inspections from a technical point of view, one should consider at least four dimensions: the inspection **process**, the inspection teams **roles**, the inspection **products**, and the used **reading techniques**.

The inspection process includes several sub-processes, as shown in figure 6.2. Depending on the chosen inspection process, some of these activities may be skipped, as noted by the optional paths presented in figure 6.2.

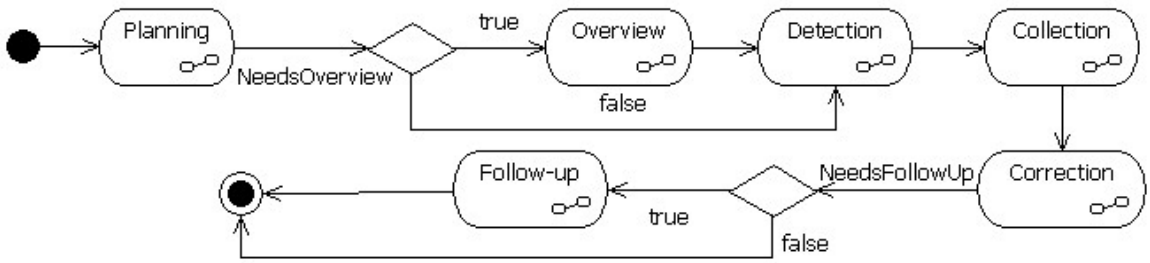


Figure 6.2: Inspection process

The **planning** activity concerns deciding which artifacts should be inspected, when, how and by whom. The **overview** activity is also known as a kick-off meeting and is used by the author of the artifact to be inspected to make a brief presentation of that artifact. The objective of this meeting is to provide a quicker familiarization of the inspection team with the artifact to be inspected, thus saving time during the next phases of the inspection. This activity is often skipped so that organizations can save the costs involved in the meeting, but is useful in situations where the briefing to be made at the kick-off is considered to be cost-effective, when compared to not holding that meeting. This can be the case when inspecting complex artifacts, or artifacts that are integrated in a complex context that the review team members are not familiar with, prior to the inspection. The defect **detection** activity is the process phase where inspectors are expected to study the inspection artifact and detect potential defects in it. Depending on the particular inspection technique being used, this can be made individually or by the inspection team, as a group. In the former alternative, the defects detected individually

have to be merged, so that the list of defects collected by all inspectors can be considered. The defect **collection** activity that follows defect detection provides the chance for reconciling the defects detected by all the inspectors into a single list, while deciding which of those alleged defects are real defects. The artifact's authors can then use the identified defects list as a checklist for the artifact's **correction**. Furthermore, the list of collected defects can be analyzed to decide whether or not it is necessary to perform a follow-up process to verify that the changes that resulted from the inspection adequately solve the identified problems.

Inspection teams have well-defined roles, which may vary from one inspection technique to another. Laitenberger's survey lists the following roles:

- the **organizer**, who is responsible for the planning sub-process;
- the **moderator**, who is responsible for leading the inspection team, particularly during meetings;
- the **inspector**, who inspects the artifact;
- the **reader**, who presents the artifact being inspected during inspection meetings;
- the **author**, who produced the artifact under scrutiny;
- the **recorder**, who keeps a log of all the defects found during an inspection meeting;
- the **collector**, who merges the defects found by all inspectors into a list, if no meeting occurs.

Note that some of these roles can be accumulated (e.g. all members participating in an inspection can act as inspectors). While some roles, such as the reader and recorder, only make sense in inspections with meetings, the role of collector is only considered when no meeting occurs. The remaining roles are usually filled regardless of the particular kind of inspection technique.

The number of inspection team members may vary, but inspection teams often have between 3 and 5 members. If the team is too small, the defect detection ratio (the percentage of defects in the artifact that are detected) may decrease. If the team is too large, the extra members will not collect a significant number of defects not detected by the other members of the team, so the extra cost of using too many inspection team members does not pay off.

There is some controversy with respect to ideal team size. Fagan recommended 4 participants as a good-sized team [Fagan 76]. Bisant and Lyle [Bisant 89] reported on significant productivity gains of novice programmers as a result of 2 person inspections and suggest such inspections can be used both in contexts where access to a larger team is not practical, and as a transition step toward using the method with

a larger team. Porter *et al.* [Porter 97c] obtained experimental results that suggest that the teams may have as little as two members, (the author and an inspector) without a significant loss of effectiveness. The number of defects reported in teams of 2 and 4 members was not significantly different. In contrast, Madachy *et al.* [Madachy 93][1] and Bourgeois [Bourgeois 96] presented data suggesting that the ideal size ranges from 3 to 5 participants.

The members of the inspection teams are usually either developers (from requirements' engineers and designers to programmers) or testers. It is usually preferable to leave managers out of the inspection process. There are at least two arguments to support this preference. On the one hand, defect detection is a technical task, and that rules out of the usefulness of involving managers, if they do not possess the adequate technical skills for the task. On the other hand, there is an important human resources management issue to consider: the artifacts, rather than their authors, are being assessed. Leaving managers out of the inspection mitigates the effects of fear from the inspection outcome. As noted by [Laitenberger 02] such fears can significantly damage the effectiveness of inspections. To illustrate this, consider a scenario where the number and severity of defects found during an inspection are used to evaluate the performance of the artifact's author. The knowledge of that evaluation criterion can bias the defect detection effectiveness. Rather than being regarded as an opportunity to, in a constructive context, increase the quality of the artifacts, inspections would then be regarded as a threat to the author's professional career.

Inspections can be conducted on several kinds of artifacts. According to Laitenberger's survey on inspections, most of the inspections reported in literature are performed on code documents. Others are performed, with a decreasing frequency, on design, requirements, and testing documents. There is no apparent relationship between the benefits of inspections and their frequency in literature, with respect to the inspection product. In general, the return on investment of defect detection techniques is considered higher when defects are detected earlier in the process [Boehm 81]. Therefore, inspections carried out in artifacts produced earlier in the process yield higher benefits, according to [Briand 98].

There are a number of reading techniques that can be used during inspections. These include, among others:

- **Ad-hoc reading**. No specific guidelines are provided to the inspectors. The inspector is free to use any strategy to uncover defects.

- **Checklist-based reading**. The inspectors are given a set of questions they have to answer, thus driving the inspector's focus of attention. This is still a non-systematic approach to defect detection, in the sense that no strategy is provided for answering the questions on the checklist [Porter 95]

[1] This work is cited in [Laitenberger 02], but we were not able to access the original paper.

- **Active design reviews**. These reviews are targeted for design artifacts and consist in assembling a team of designers with varied expertise, so that each can focus his attention in particular parts of a design. Inspectors actively review the design under scrutiny by answering a set of questions created to foster a deep understanding of the design, thus preventing reviewers from just skimming the artifact [Parnas 85].

- **Defect-based reading**. In this technique, inspectors focus on different defect classes, while inspecting the artifacts. Each defect class has its own set of questions that the inspector should answer [Miller 98].

- **Perspective-based reading**. In this reading technique, each reviewer focuses on the point of view of one of the customers of the inspected artifact. These customers may be testers, developers, and so on. The rationale is that different perspectives will tend to drive the focus of attention of reviewers to different aspects of the artifact, thus increasing the coverage of the inspection, while each inspector is concerned with a narrower and deeper scrutiny of the inspected artifact [Basili 96b].

- **Reading by stepwise abstraction** [Dyer 92a] is an inspection technique applied to code, where the inspector reads a set of instructions and abstracts these instructions compute, repeating this process until the artifact is abstracted to the point where it can be compared with its specification. This technique is used in conjunction with the **cleanroom approach** [Dyer 92b].

6.2.2 Inspection success drivers

Understanding what drives inspections' success has been a long time concern in the software community. Based on data collected from over 6000 inspections, Weller studied the impact of the inspection process on software quality [Weller 93]. Among several other remarks, he pointed to the familiarity of the inspection team with the artifact being inspected as a key factor in inspection success. We may regard this as kind of **domain expertise**.

Siy observed that while structural changes were largely ineffective in improving the results of inspections, the **inputs** for those inspections (the reviewers and code being inspected) were far more influential in the inspection outcome [Siy 96]. These findings were further explored in [Porter 98], to conclude that better inspection techniques, rather than processes, were the key to improving inspection effectiveness. Biffl and Halling combined reviewers' expertise measures (software development skills, experience and an inspection capability pre-test) with different code inspection techniques [Biffl 02]. While they could not find significant relationships between development skills and experience and inspectors' performance, they found the inspection

capability pre-test useful to optimize the inspection outcome by selecting ideal inspection teams. They also identified performance differences related to alternative code reading techniques, a result that is consistent with the findings of Laitenberger and DeBaud, in their systematic review on code inspections reading techniques [Laitenberger 00]. Sauer *et al.* identified individual's task expertise as the primary driver of review performance [Sauer 00].

In a totally different context (social psychology), Kruger and Dunning observed that the skill of a person in performing a task is closely related to the required skill to assess his own performance in the same task [Kruger 99]. If we instantiate this insight into code production and code reviewing, we would expect the best programmers to also be the most effective code reviewers, although this intuition was not confirmed in Biffl and Halling's paper [Biffl 02].

6.3 Experimental planning

6.3.1 Goals

The goal presented in section 6.1.2 is too abstract for the purposes of our assessment. To make it more concrete, we use the expected expertise impacts, as presented in figure 6.1, as a reference and break our abstract goal into more concrete sub-goals. In the following sub-goals definition "(...)" is used to denote that we keep the corresponding part of the more abstract goal definition. This allows us to highlight the differences among the four sub-goals.

Goal 1(G1):
Analyze the outcome of software components source code inspections,
(...)
with respect to the impact of developer's skill on defect detection,
(...)

Goal 2(G2):
Analyze the outcome of software components source code inspections,
(...)
with respect to the impact of peer's skill on defect detection,
(...)

Goal 3(G3):
Analyze the outcome of software components source code inspections,
(...)
with respect to the impact of reviewer's (developers and peers) skill on defect detection,

(...)

Goal 4(G4):
Analyze the outcome of software components source code inspections,
(...)
with respect to the impact of the gap of skill between developers and peers on defect detection,
(...)

6.3.2 Experimental units

This experiment occurred in the context of a Software Engineering course held at the Universidade Nova de Lisboa, during the Spring semester of 2005. This course is offered on the 8th semester of their 5-years informatics degree. For reference, the adopted instantiation of the Bologna process at the informatics course reserved 3 years for the first cycle, and 2 for the second one. The participants are comparable to MSc. students.

The participants were grouped into teams. As there was an odd number of participants, one team had three participants. All the remaining teams had two members. There were a total of four different development tasks. Each task consisted on developing a component, or set of components. The component-based system was built from the assembly of the deliverables of these four development tasks.

Each development team was randomly assigned to one of these tasks. A set of four development teams would then perform the final integration project (thus evolving the test bed, if necessary, from an application prototype to the final project). The selection of the combinations of four teams that performed the final integration project was performed **after** the code inspections carried out in the experiment described here. It had no influence in the outcome of the inspections.

With respect to the selection of peer teams that, along with the developers, made up the inspection teams, the peer teams were randomly selected from the set of teams which were performing a different development task. In practice this means that the peers were familiar with the requirements of the software they were to inspect, but had not developed software with the same functionality themselves. So, for instance, if the source code of component X was to be inspected, then the peer team would have to be chosen from those who were not developing component X. This emulates the situation in a professional environment, where peers are developers not involved in the development of the artifact being inspected, but are otherwise familiar with both the development techniques and the requirements of the artifact. The random assignment of reviewers also rules out the possibility of participants choosing the teams whose artifacts they would inspect.

6.3.3 Experimental material

The course's project consisted in developing a component-based elevator system simulator from requirements definition to final product delivery. As explained in the previous sub-section, the project was divided into four sub-projects that were to be integrated in the end. For illustration purposes, figure 6.3 presents the four development tasks (**Motor & Alarm**, **Elevator Controller**, **Request Manager**, and **Test Bed**). Three of the tasks consisted on developing the components to be used in the final system, while the forth (Test Bed task) consisted on developing the component-based system, by creating mock-up implementations of the four major components of the system. The details of this particular component-based architecture are not relevant for the discussion in this dissertation.

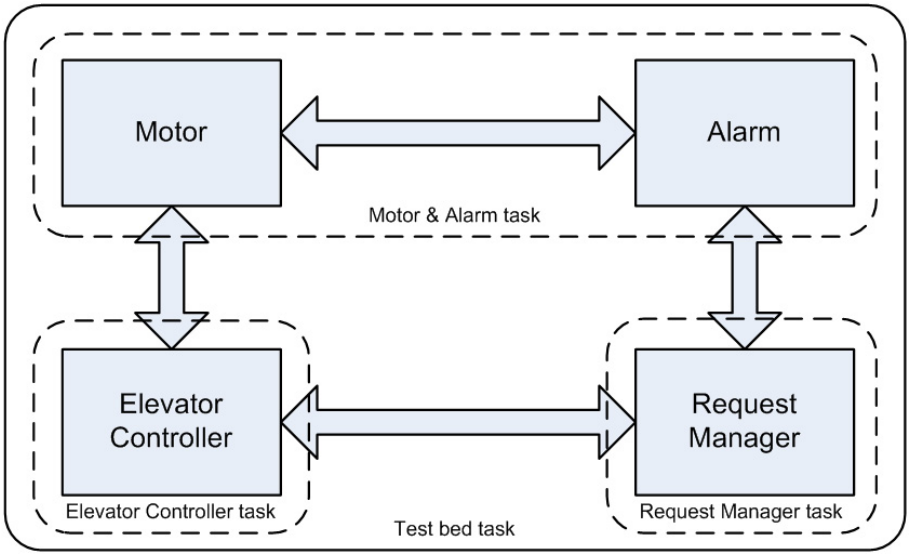


Figure 6.3: Development tasks in the elevator project

The overall development process followed in the project is depicted in figure 6.4. The activity under scrutiny in this experiment was the code inspection carried out exactly once for each component, in step 5.

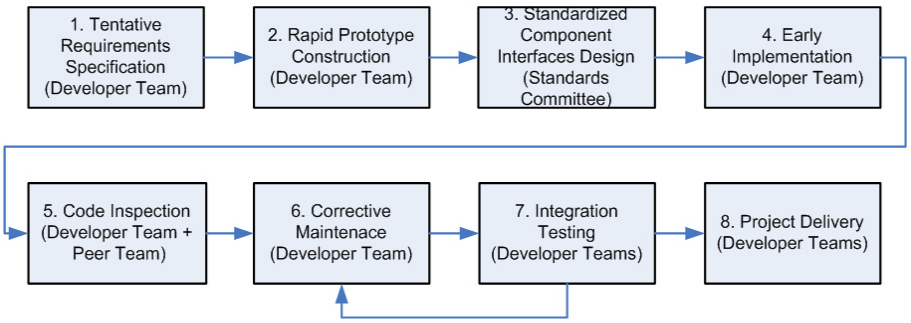


Figure 6.4: Development process in the elevator project

The specific combinations of integration teams were unknown to the participants at the time this experiment was performed (they were only revealed to participants

when starting step 7, in the development process). The secrecy of the used combinations of components for integration attempted to simulate the situation in a component development environment where the set of clients of the components are at least partially unknown to the component producers. This is often the case when developing off-the-shelf components for the component marketplace.

The programming language used during development was Java, well-known to all subjects in the experiment, who started programming in Java in the third semester of the course. The inspections carried out during this experiment were performed on the software components developed during the course's project. Each component consisted on a set of (frequently multi-threaded) classes that, as a whole, implemented a well-defined interface, specified by the participants in the third activity of the development process depicted in figure 6.4. Although the overall example of the elevator system simulator can be considered a toy example, the dimension and complexity of the inspected source code is comparable to the one that would be used in code inspections conducted in a professional environment. Note that time constraints limit the size and complexity of artifacts that can be assessed, even in a professional environment, in a code inspection.

6.3.4 Tasks

Among other activities, the development process included an inspection performed on all the developed components. The planning of the inspection was performed by the course's tutors.

The entry criteria for components to be inspected was that their source code corresponded to an early version of the component which was expected to be fully functional, by the time the component was inspected. Note that, while encouraged, the full functionality of the component could not be verified *a priori* (this verification was one of the expected outcomes of the inspection process). With respect to participants selection, the selection was stipulated by the course's tutors, in a process that joined a development team with a peer teem, to build the full inspection team. The inspection teams were built in such a way that although peer team members were knowledgeable in the inspected code basic requirements, they were not developing an alternative implementation of that same component, to avoid biasing their review with their own experience in the project. Note that each participant was asked to contribute to two different code inspections: one, where he was a member of the development team, and another where he acted as a peer team member.

In any given inspection, the following inspection roles were assigned to the four review team members: the development team members got the **moderator** and **author** roles and the peer team members the remaining ones (**reader** and **recorder**). All acted as inspectors. Assigning one of the authors as the moderator of the inspection meeting is not common practice. However, as there were two developers rather than just one

we had to assign one of the developers to a role which is normally performed by a peer. Considering the available alternatives (moderator, reader, and recorder) and the context under which the experiment took place, we decided that the moderator's role was the most innocuous for this adaptation, as we wanted to ensure that neither the reading of the artifact nor the recording of the review data would be performed by an author. Having a peer paraphrasing the code can help uncovering portions of the code that are unclear for someone other than the code's author. It also denies the author the possibility of skimming through parts of the code he is not particularly confident about, fearing he is being indirectly assessed. With respect to the recorder, we wanted to avoid an incomplete or less detailed recording of problems that could result both from fear of poor evaluation and from knowledge which could be assumed to be tacit by a recorder who was a developer of the artifact, as well.

The optional overview meeting was skipped because the requirements of each of the inspected components were well-know to all participants, prior to the inspection, as they were thoroughly discussed earlier in the course (development tasks 1 through 3).

All inspectors were instructed to make a solo review of the articles before the meeting. This review had no strict time limits and participants were asked to record its actual duration. As this solo review was conducted off-line, there is no way to validate the accuracy of the recorded preparation time, so, we decided against using this variable as a predictor in our models. The average preparation time for inspection meetings was around 1 hour, as suggested to participants during training.

With respect to the used reading technique, an extensive checklist of common defects in Java programs was distributed (and its contents explained) to all review teams before code inspections (including the solo reviews) took place. Checklist-based reading is one of the two reading techniques that do not require special training and have been validated through industrial practice [Laitenberger 02]. When compared to the other technique that fills these two criteria (ad-hoc reading), checklists offer some level of support for defect detection, as opposed to none. The availability of a common defects taxonomy facilitates the comparison of the outcomes of all inspections conducted in this experiment. Other, more sophisticated, reading techniques could also have been tried, but they would have at least three drawbacks for our purposes in this experiment:

- In the context where this experiment took place, the need for special training in a particular reading technique was not easily accommodated within the course's time constraints.

- Sophisticated reading techniques are not common industrial practice, unlike checklist-based, or ad-hoc reading. This would be a threat to the validity of results in industrial settings.

- Sophisticated reading techniques try to, among other things, offer guidance to inspectors that is expected to help them to detect defects more effectively, and, if possible, in a repeatable way. A side effect, with respect to our experiment, is that the more successful these techniques are at guiding inspectors, the less visible the effect of individual expertise becomes.

Defect collection was performed during an inspection meeting, where the whole inspection team followed the readers' presentation of the inspected code and identified the potential defects. The teams reached a consensus with respect to which of the identified potential defects were really defects, and the recorder filled a special defect detection report form prepared by the tutors to facilitate defect collection in a format which would be also convenient to facilitate further analysis.

With respect to follow-up, the development teams were responsible not only for fixing the identified problems, but also for filling a follow-up form.

6.3.5 Hypotheses and variables

Hypotheses

The observations on the problem statement section lead us to testing four different basic hypotheses, to assess the effect of practitioners' expertise on the outcome of the code inspection, in terms of the inspected defects diversity. We identify the hypotheses as $H1$, $H2$, $H3$, and $H4$. For each of them, we formulate both a null and an alternative hypothesis (e.g. $H1_0$ and $H1_1$).

Our research hypotheses are as follows:
$H1_0$: Developer skill has no effect on the inspected defect diversity.
$H1_1$: Developer skill has an effect on the inspected defect diversity.
$H2_0$: Peer skill has no effect on the inspected defect diversity.
$H2_1$: Peer skill has an effect on the inspected defect diversity.
$H3_0$: Reviewer expertise has no effect on the inspected defect diversity.
$H3_1$: Reviewer expertise has an effect on the inspected defect diversity.
$H4_0$: The gap of expertise between developer and peer has no effect on the inspected defect diversity.
$H4_1$: The gap of expertise between developer and peer has an effect on the inspected defect diversity.

Independent variables

The basic independent variable of this experiment is the *subjects' expertise*. We use two measures of our subject's expertise: their ***A**verage **G**rade* (*AG*) throughout their academic path, based on the independent evaluation our subjects received in over 30 different

courses, and the ***N**umber of **Sem**esters* (*NSem*) it took them to complete those courses. We assume that there is a higher merit in obtaining a given *AG* in the ***R**ecommended number of **Sem**esters* (*RSem*), than in a higher *NSem*. The ***S**imple **W**eighted **A**verage **G**rade* (*SWAG*) and the ***C**omplex **W**eighted **A**verage **G**rade* (*CWAG*) expertise metrics, defined below, follow this rationale.

$$SWAG = AG \times \frac{RSem}{Max(RSem, NSem)}$$

$$CWAG = AG \times \sqrt{\frac{RSem}{Max(RSem, NSem)}}$$

Note that *SWAG* causes a bigger penalty than *CWAG*, as *NSem* increases. Figure 6.5 shows how the penalty for loosing semesters affects the mean classification of a student, with our merit assessment scheme. For instance, a student who has lost a year before enrolling to this course would have a penalty resulting from the two extra semesters. The penalty factor would then be multiplied by the student's average grade, to obtain the student's merit factor. So, for a student who enrolled in this course at his 10^{th} semester, rather than at his 8^{th}, with an average grade of 15, we would have $AG = 15$, $SWAG = 12$, and $CWAG = 13,42$.

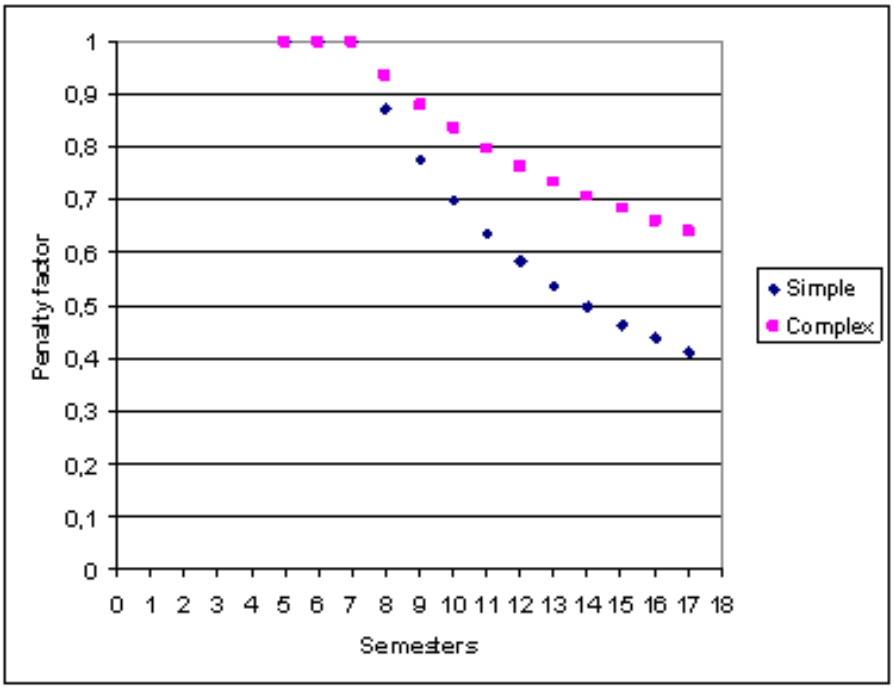


Figure 6.5: Subjects expertise penalty factors, with respect to the number of extra semesters taken before participating in this experiment.

Hypotheses H_1 through H_4 use the expertise of different teams. In hypotheses H_1, H_2, and H_3, we use the *best subject*, the *worst subject*, and the *average within the team*. Finally, in hypothesis H_4 we consider the difference between the expertise of the development team and that of the peer team as an independent variable. The rationale for

using the mean, worst, and best expertise assessments for each team is that we do not know *a priori* what the effect of different expertise levels, within each team will have on our overall evaluation. In summary, we have 3 alternative rating schemes for grades, and 3 ways of combining grades within teams. This implies that we have 9 different ways for quantifying our independent variable (the expertise). These alternatives are used for each hypothesis under test.

Table 6.1 presents a condensed view of the independent variables tested in this experiment, as predictors of the diversity of errors found during code inspections. The first row of labels indicates the hypothesis which the independent variable will be used to test. The second row of labels represents the considered teams. The column labels indicate if we are using the mean, worst, or best elements of the teams in our as independent variable. The inner table cells refer to the specific metric of expertise used. For example, if we want to test H_4 using the mean difference of expertise between the development team and the peer team, with *SWAG* as the expertise metric, our independent variable is $A\_Diff\_DT\_PT\_SWAG$, presented in **bold** in the table.

	H1			H2			H3			**H4**		
	DT			PT			RT			**RT_Diff**		
A	AG	SWAG	CWAG	AG	SWAG	CWAG	AG	SWAG	CWAG	AG	**SWAG**	CWAG
W	AG	SWAG	CWAG	AG	SWAG	CWAG	AG	SWAG	CWAG	AG	SWAG	CWAG
B	AG	SWAG	CWAG	AG	SWAG	CWAG	AG	SWAG	CWAG	AG	SWAG	CWAG

Table 6.1: Independent variables

Dependent variables

The dependent variables used in this experiment represent the diversity of defects found during code inspection. A defect classification checklist was distributed to all participants. The checklist contained 16 different defect classes, which were then subdivided into a total of 81 different defect codes. In summary, our dependent variables are:

- *NDSCode* = the number of different specific defect codes reported in the inspection
- *NDGClass* = the number of different generic defect classes reported in the inspection

At first, *NDGClass* may seem unnecessary, given the usage of a finer grained measure (*NDSCode*). However, if two code inspections report a similar number of different defect codes, but one of them uses a lot less defect classes than the other, it may be the case that this reflects a lower coverage of the kinds of problems to be found during the inspection. We used *NDGClass* to detect this kind of problem, should it occur.

6.3.6 Design

The design used in this experiment can be classified as a **quantified multiple control groups post-test only design**, which can be presented as follows, using Trochim's notation [Trochim 06][2]:

```
Group Q1 C1 X O
Group Q2 C2 X O
Group Q3 C3 X O
Group Q4 C4 X O
```

This design results in 4 groups, from Q1 through Q4 that are selected using a cut-off method (denoted by **C1** through **C4**). The cut-off criteria used in our experiment is the division of the original sample into quartiles, using an expertise assessment metric as the discriminant value to assign each inspection to the proper quartile. Groups Q1 through Q4 are ordered by the expertise assessment metric.

All groups receive a similar treatment X (the code inspection) followed by a similar observation O (the inspection reports). Note that while performing the experiment, subjects are completely unaware of their assignment to these groups. As far as they are aware, all participants are treated equally. For the purposes of our assessment, we can consider that the treatments were administered at the same moment in time, as the time gap between all the inspections is insignificant in our context.

It should be noted that although the inspection on a given artifact is only conducted once, as we have several alternative expertise metrics and two alternative inspection outcome metrics, this design is replicated for each of the possible combinations, for the purpose of subsequent data analysis.

6.3.7 Procedure

Regarding the experiment instrumentation, the calculation of subjects' expertise was done upon the data available from the university's academic database. The information concerning code inspections was collected from the standard inspection reports submitted by subjects after they performed the code inspections. The information collected through these procedures can be represented by instantiating the class diagram presented in figure 6.6.

With respect to the definition of the metrics outlined in section 6.3.5 to support the independent and dependent variables, this definition can be formalized in OCL, upon the model presented in figure 6.6. This is a variation on the technique presented in earlier chapters, in the sense that we are using a model for the definition of the met-

[2]In chapter 3 we presented a class diagram (figure 3.11) that could be instantiated here to represent our experimental design. While storing the design information by instantiating those classes is practical for further computer supported analysis, the resulting object diagram is cumbersome. For visualization purposes, we will maintain Trochim's notation.

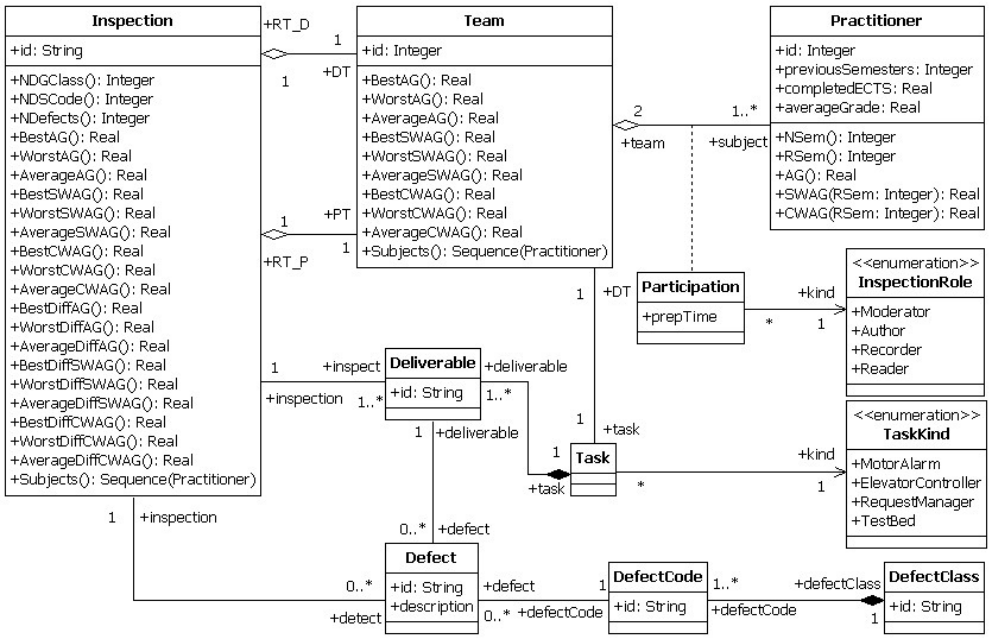


Figure 6.6: Experiment data class diagram

rics, rather than a metamodel. So, rather than instantiating a metamodel on software inspections with a model of our inspection, we are defining a model of the code inspection carried out in our experiment in UML, and then using OCL to specify metrics upon the model. So, rather than Metamodel Driven Measurement, we are using Model Driven Measurement in this chapter.

The class diagram in figure 6.6 includes the signatures of the operations added to our model classes to support metrics definitions.

The operations defined for the class `Practitioner` are presented in listing 6.1. `NSem`, `RSem`, and `AG` are trivial. In the definition of `SWAG` and `CWAG`, we use two auxiliary functions, `maxPair`, `minPair` and `sqrt`, which are not part of the OCL standard. `maxPair` returns the maximum value between its two arguments, `minPair` returns the minimum, and `sqrt` returns the squared root of its argument.

Listing 6.1: Practitioner metrics in UML 2.0.

```
context Practitioner
  NSem(): Integer = previousSemesters
  RSem(): Integer = requiredSemesters

  AG(): Real = averageGrade
  SWAG(): Real = AG() * (RSem()/(maxPair(RSem(), NSem())))
  CWAG(): Real = AG() * sqrt(RSem()/(maxPair(RSem(), NSem())))
```

The operations defined for the class `Team` are presented in listing 6.2. A `Team` has a set of practitioners (`subject`), and its metrics are computed upon those practitioners. The collection of practitioners is obtained through the `Subjects()` operation, which returns the collection as a sequence of practitioners. Note that `Team` refers to either a

development team or to a peer team. For each of the metrics **AG**, **SWAG**, and **CWAG**, the best, worst and average values are computed. In these formalizations, the existence of two operations which are not part of the standard OCL is also assumed. The operation `max()` receives a sequence of real numbers and returns the maximum value in that sequence. The operation `min()` is similar to the former, but returns the minimum value.

Listing 6.2: Team metrics in UML 2.0.

```
context Team
  BestAG(): Real = max(Subjects().AG())
  WorstAG(): Real = min(Subjects().AG())
  AverageAG(): Real = Subjects().AG()->sum()/Subjects()->size()

  BestSWAG(): Real = max(Subjects().SWAG())
  WorstSWAG(): Real = max(Subjects().SWAG())
  AverageSWAG(): Real = Subjects().SWAG()->sum()/Subjects()->size()

  BestCWAG(): Real = max(Subjects().CWAG())
  WorstCWAG(): Real = min(Subjects().CWAG())
  AverageCWAG(): Real = Subjects().CWAG()->sum()/Subjects()->size()

  Subjects(): Sequence(Practitioner) = subject->asSequence
```

Finally, the `Inspection` class has the links to the development team `DT` and peer team `PT` that, as a whole, form the inspection team. As such, the `Subjects()` operation performs the union of the subjects in the development and peer teams. The operations for computing the dependent variables `NDGClass`, `NDSCode` and `NDefects` count the number of defect classes, codes, and defects, respectively, found in each inspection. The independent variables are computed through operations that have definitions similar to those of the class `Team`. However, these values are now computed for different collections of practitioners (i.e. the whole inspection team).

Listing 6.3: Team metrics in UML 2.0.

```
context Inspection
  -- Operations for computing the dependent variables
  NDGClass(): Integer =
    self.detect->collect(defectCode)->asSet()->
      collect(defectClass)->asSet()->size()
  NDSCode(): Integer = self.detect->collect(defectCode)->asSet()->size()
  NDefects(): Integer = self.detect->size()

  -- Independent variables
  BestAG(): Real = max(Subjects().AG())
  WorstAG(): Real = min(Subjects().AG())
  AverageAG(): Real = Subjects().AG()->sum()/Subjects()->size()

  BestSWAG(): Real = max(Subjects().SWAG())
```

```
  WorstSWAG(): Real = max(Subjects().SWAG())
  AverageSWAG(): Real = Subjects().SWAG()->sum()/Subjects()->size()

  BestCWAG(): Real = max(Subjects().CWAG())
  WorstCWAG(): Real = min(Subjects().CWAG())
  AverageCWAG(): Real = Subjects().CWAG()->sum()/Subjects()->size()

  BestDiffAG(): Real = DT.BestAG() - PT.BestAG()
  WorstDiffAG(): Real = DT.WorstAG() - PT.WorstAG()
  AverageDiffAG(): Real = DT.AverageAG() - PT.AverageAG()

  BestDiffSWAG(): Real = DT.BestSWAG() - PT.BestSWAG()
  WorstDiffSWAG(): Real = DT.WorstSWAG() - PT.WorstSWAG()
  AverageDiffSWAG(): Real = DT.AverageSWAG() - PT.AverageSWAG()

  BestDiffCWAG(): Real = DT.BestCWAG() - PT.BestCWAG()
  WorstDiffCWAG(): Real = DT.WorstCWAG() - PT.WorstCWAG()
  AverageDiffCWAG(): Real = DT.AverageCWAG() - PT.AverageCWAG()

  Subjects(): Sequence(Practitioner) = DT.Subjects()->union(PT.Subjects())
```

As mentioned earlier, in listings 6.1 through 6.3 we used several utility functions that are not part of the OCL standard. One possible way of implementing those operations is to create a library class that we will call `Utils`, and then define these operations as class operations. As this would clutter slightly the presented definitions, and is not supported by the OCL tool used throughout the preparation of this dissertation, another implementation alternative is to make `Inspection`, `Team`, and `Practitioner` subclasses of our `Utils` class. A possible implementation of these operations would be the one in listing 6.4.

`sqrt()` is implemented following the Newton's iteration algorithm [3] Although the default number of iterations set here is 5, for illustration purposes, this number could be changed to adjust the desired precision of the algorithm. Although OCL provides a convenient way of accessing the first element of a sequence, through the selector `first()`, it does not provide an utility operation for the remaining ones. We define such an operation (`tail`) in our utilities library. Both `max` and `min` have a pre-condition: the sequence `s` of numbers cannot be empty (`s->size()>0`). These functions are specified in listing 6.4.

Listing 6.4: Team metrics in UML 2.0.

```
context Utils
  sqrtAux(r: Real, itCount: Integer): Real =
    if (itCount > 0)
    then (0.5 * (sqrtAux(r, itCount-1) + (r/sqrtAux(r, itCount-1))))
    else 1.0
```

[3] For further details on the Newton's iteration algorithm, see, for instance: `http://mathworld.wolfram.com/NewtonsIteration.html`

```
  endif

sqrt(r: Real): Real = sqrtAux(r, 5)

maxPair(a: Real, b: Real): Real = if ( a >= b ) then a else b endif
minPair(a: Real, b: Real): Real = if ( a <= b ) then a else b endif

tail(s: Sequence(Real)): Sequence(Real) =
  if (s->isEmpty())
  then s
  else s->excluding(s->first)
  endif

max(s: Sequence(Real)): Real =
  if (s->size()>1)
  then maxPair(s->first(), max(tail(s)))
  else s->first()
  endif

min(s: Sequence(Real)): Real =
  if (s->size()>1)
  then minPair(s->first(), min(tail(s)))
  else s->first()
  endif
```

6.3.8 Analysis procedure

Data analysis is to be carried out through the following steps:

- **Descriptive statistics**: For all our independent and dependent variables, we will collect a set of descriptive statistics including the **mean**, **standard deviation**, the **minimum** value of the variable in the sample, the **maximum** value, as well as the skewness (a measure of the asymmetry of the distribution of the variable) and the kurtosis (a measure of the "peakidness" of the distribution of the variable - higher kurtosis occurs when the variance in the sample is due to infrequent extreme deviations, while a lower one corresponds to smaller frequent deviations) of the variable's distribution. These descriptive statistics will provide us with a first overview on our data, that we will further detail in subsequent analysis.

- **Data set reduction**: in a data distribution, the presence of outlier and extreme values can change our view on the relations between dependent and independent variables. Therefore, before proceeding with further tests, we need to determine whether or not these values occur in our data.

- **Normality tests**: Before engaging into statistical tests to verify our hypotheses, we first have to check our data's distribution. This is important, so that we can

select statistical tests that are adequate for our data. In particular, normality tests will allow us to decide whether we should use parametric tests, or non-parametric ones. The former are, in general, more powerful than the latter, but require the data distribution to be known. The latter can be used if the normality tests show that our data does not have a normal distribution. In any case, further tests to the data may follow, to ensure the results of the statistical tests used are meaningful.

- **Correlation analysis**: These tests will allow us to verify if there is a statistically significant relation between our independent and dependent variable. If so, the dependent variable can be regarded as a sign of the independent one, and we will further explore the relationship. Otherwise, we will conclude that, as no correlation exists, the hypothesis under scrutiny can be rejected at this point.

- **Analysis of differences between groups**: Finally, for the hypotheses that were not eliminated in the previous step, we will perform a test to detect whether there are significant differences between groups (when compared to differences within those groups). In other words, we will find out whether any of the groups created in our experimental design exhibits a significantly different behavior, when compared to the others. If so, we will also determine if there are noticeable trends, from one group to the next.

We will detail each of these steps in section 6.5 and provide some insight into the specific statistics tests, with an emphasis on their interpretation, as we use them.

6.4 Execution

6.4.1 Sample

In the beginning of the semester, there were 93 students enrolled in the course. Five of them dropped out before the experiment started, and one also gave up before turning in the first implementation of his group's component. The remaining 87 students completed the project and are the subjects of this experiment. They were paired into 44 development teams. 43 of those DTs produced components that were inspected. The deliverables of these 43 inspections were used to collect the dependent variables.

As stated before, the participants in the experiment were students enrolled in the 8^{th} semester of the informatics degree at Universidade Nova de Lisboa. Our subjects had a mean of *198,3* ECTS credits (with a standard deviation of *30,3*) before participating in this experiment. For reference, accomplishing the 1^{st} cycle (BSc degree) requires *180* ECTS, while *300* ECTS are required for obtaining a MSc degree.

The *87* subjects participating in this experiment are a convenient, but also representative sample of the informatics students which annually graduate from Universidade

Nova de Lisboa. At the time of the experiment, the *numerus clausus* for the informatics degree was *160*, and the number of students graduating each year was around *60*.

6.4.2 Preparation

The subjects were not aware of the aspects being researched, at the time they participated in the experiment, as this could jeopardize the validity of the results. They were only aware of our intention to use data collected during the project (from any project phase, rather than from the code inspections, in particular).

Prior to the implementation of the components that were later inspected, subjects received a Java coding style guide, along with a specification of the provided interfaces of each component. The interfaces were specified as Java interfaces, that had to be implemented by the corresponding components.

Subjects received training on how to perform code inspections, before actually starting them. Then, a week before the inspection meeting, the review teams received the code they would have to inspect and were instructed to study it, in order to prepare the inspection meeting. Subjects also received a code defects checklist to support the inspection. Although this was not enforced or verifiable, it was suggested that the expected meeting preparation time should be about 1 hour. The inspection package received for the preparation also included the inspection report template that would be used to record all the defects found in a standardized way.

6.4.3 Data collection performed

The experimental process was not allowed to disturb in any way the subjects' activities in the project. Subjects performed their normal tasks while developing this project, from requirements specification down to project delivery. In this sense, code inspections were regarded as yet another one of the project's activities, rather than receiving a particular focus, from the participants point of view. Code inspection data was collected from the project's deliverables, which was checked-in in a contents management system made available to participants. Inspection meetings typically lasted for about one hour. Each team was involved in two inspections. In one of the inspections, they acted as the developers within the review team. In the other inspection, they were the peer members in the review team. With few exceptions, both inspections were carried out consecutively (in a two-hours class). The total set of inspections was conducted in two consecutive days.

6.5 Analysis

6.5.1 Descriptive statistics

The descriptive statistics for our independent and dependent variables are summarized in table 6.2. We will not discuss in detail each of the variables, for the sake of brevity. Instead, we discuss with some more detail, for illustration purposes, two of these variables, *NDSCode*, and *B_DT_AG*. To help in this analysis, we also include figures 6.7 and 6.8, where we can observe an histogram for the distribution of the *NDSCode* variable and a boxplot of the distribution of *B_DT_AG*.

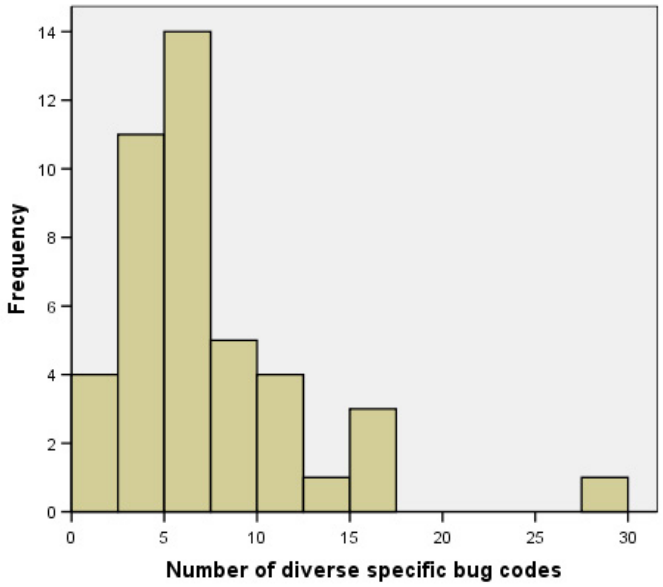


Figure 6.7: NDSCode histogram

The histogram in figure 6.7 graphically presents the distribution of the number of defect classes found in the inspections. The most noteworthy case is the one where *28* different defect codes were registered, in a sample where the mean number of different defect codes is around *7*. Note also how, at a first glance, the distribution seems to be right-skewed (as is confirmed by its positive skewness value, *2,025*, in table 6.2. This denotes a higher concentration of values on the right tail of the distribution than would be expected in a normal distribution. The high kurtosis value (*5,702*) hints that the distribution has a wider spread than what is to be expected from a normal distribution. In other words, we have more observations at the extremes than what would be expected in a normal distribution.

The boxplot in figure 6.8 presents a different view on the distribution of a variable, in this case the *B_DT_AG* (developer team student's mean grade), where the lower

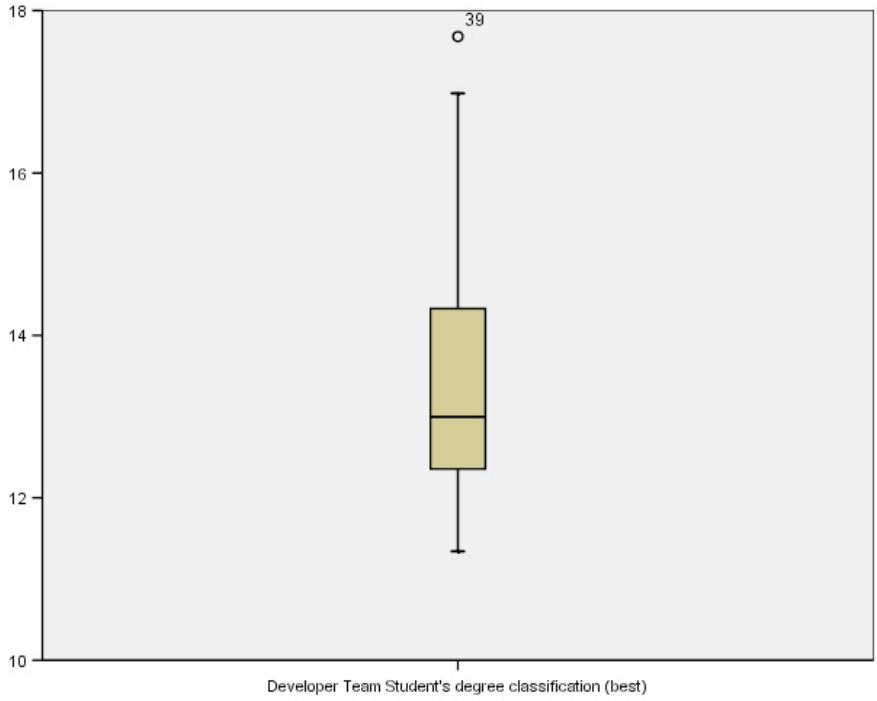


Figure 6.8: B_DT_AG boxplot

whisker, lower, median, and upper quartiles, and upper whisker are represented as horizontal lines. They represent, respectively, the lower value (excluding outliers), the lower quartile (which corresponds to the value that cuts off lowest *25%* of data), the median (*50%*), upper quartile (*75%*), and the upper value (again, excluding outliers). An example of an outlier is denoted by the circle marked with the case id number *39*. We can observe that this expertise assessment of the teams show that there is a higher concentration of teams in the lower quartiles and their dispersion increases as we move to higher ones. This not only hints a non normal distribution, as we will confirm later, but is also helpful in the interpretation of the statistical tests we will perform to assess our hypotheses. The mean value for this metric is *13,409* (in a scale from *0,000* to *20,000*), the minimum is *11,340* (about *2* points away from the mean) and the maximum is *17,680* (more than *4* points away from the mean). In short, there is a smaller difference between the worse and average students in our sample, than between the average and the best students, with respect to this expertise metric. With small variations, this pattern is repeated in most of the expertise assessments in our sample, suggesting that we have a Paretto distribution of expertise that contrasts the "vital few" best students with the "trivial many" remaining students. This tendency is mostly noticeable when using the plain *_AG_* metrics. Both the *_SWAG_* and the *_CWAG_* metrics mitigate this effect through the introduction of the element of the number of semesters into their equations.

Note also that the *_Diff_* metrics on table 6.2 follow a different pattern, because rather than expressing directly the expertise, they express the expertise gap between developer teams and their peers in code inspections. So, unlike direct expertise assess-

ments, these metrics have a mean value centered on *0* and, as shown in table 6.3 have a normal distribution.

	Mean	Std. Dev.	Minimum	Maximum	Skewness	Kurtosis
NDSCode	7,093	5,117	1,000	28,000	2,025	5,702
NDGClass	5,047	2,681	1,000	12,000	0,896	0,448
A_DT_AG	13,041	1,103	11,330	15,480	0,701	-0,629
W_DT_AG	12,673	0,997	11,320	14,750	0,667	-0,656
B_DT_AG	13,409	1,339	11,340	17,680	1,223	1,728
A_DT_SWAG	10,155	3,209	3,954	15,480	0,128	-1,144
W_DT_SWAG	9,272	3,450	2,937	14,750	0,119	-1,148
B_DT_SWAG	11,038	3,278	4,970	17,680	0,102	-1,135
A_DT_CWAG	11,386	2,218	7,224	15,480	0,266	-0,812
W_DT_CWAG	10,716	2,346	5,769	14,750	0,078	-0,626
B_DT_CWAG	12,045	2,323	8,157	17,680	0,443	-0,518
A_PT_AG	13,017	1,085	11,330	15,480	0,756	-0,459
W_PT_AG	12,671	0,973	11,320	14,750	0,660	-0,503
B_PT_AG	13,363	1,324	11,340	17,680	1,346	2,127
A_PT_SWAG	10,151	3,102	3,954	15,480	0,142	-1,022
W_PT_SWAG	9,287	3,293	2,937	14,750	0,133	-0,995
B_PT_SWAG	11,014	3,232	4,970	17,680	0,110	-1,071
A_PT_CWAG	11,384	2,145	7,224	15,480	0,280	-0,639
W_PT_CWAG	10,733	2,256	5,769	14,750	0,047	-0,398
B_PT_CWAG	12,023	2,277	8,157	17,680	0,479	-0,357
A_RT_AG	13,029	0,742	11,753	15,168	0,458	0,296
W_RT_AG	12,125	0,578	11,320	13,800	0,835	0,634
B_RT_AG	14,120	1,356	12,070	17,680	1,009	1,066
A_RT_SWAG	10,153	2,341	5,155	15,168	0,005	-0,414
W_RT_SWAG	7,432	2,688	2,937	13,280	0,507	-0,164
B_RT_SWAG	12,917	2,775	6,585	17,680	-0,513	-0,345
A_RT_CWAG	11,385	1,585	7,943	15,168	0,009	-0,135
W_RT_CWAG	9,453	1,778	5,769	13,280	-0,116	0,354
B_RT_CWAG	13,384	2,116	8,974	17,680	-0,017	-0,317
A_RT_Diff_AG	0,024	1,607	-3,525	3,135	-0,159	-0,206
W_RT_Diff_AG	0,002	1,455	-3,250	3,060	-0,130	-0,150
B_RT_Diff_AG	0,046	1,921	-4,890	5,020	-0,109	0,563
A_RT_Diff_SWAG	0,004	4,233	-9,783	8,013	-0,055	-0,510
W_RT_Diff_SWAG	-0,016	4,436	-11,630	7,938	-0,327	-0,262
B_RT_Diff_SWAG	0,024	4,521	-7,936	9,151	0,128	-0,857
A_RT_Diff_CWAG	0,002	2,998	-7,453	5,394	-0,170	-0,391
W_RT_Diff_CWAG	-0,017	3,098	-8,796	5,544	-0,479	0,124
B_RT_Diff_CWAG	0,021	3,235	-6,110	6,311	0,115	-0,854

Table 6.2: Descriptive statistics

In order to decide whether or not we can use parametric tests in our subsequent analysis, we have to test these variable's distributions with respect to their distribution. Parametric tests are more powerful than non-parametric ones, but require the variable's distribution to be known. The most commonly used distribution for this purpose is the normal distribution.

There are several tests which can be used to test for normality. The **Kolmogorov-Smirnov with the Lilliefors correction** test is the most widely used, according to [Maroco 03], although for relatively small samples, the Shapiro-Wilk test is considered preferable. The actual threshold of what is considered a *"small sample"* is not consis-

tently reported in the literature[4], so we used a Kolmogorov-Smirnov with the Lilliefors correction test, to determine whether our dependent and independent variables have a normal distribution and confirmed this assessment with the Shapiro-Wilk test, as well.

Table 6.3 presents these normality tests, for all the identified variables. In both tests, the normality hypothesis can not be rejected if the significance of the test is less than *0,05*. In other words, if the variable's normality test has a significance level (p-value) greater than *0,05*, we can assume the variables' distribution to be normal, with an error probability of *5%*. The non-normal variables (according to at least one of the normality tests) are highlighted in bold, in table 6.3, as is the test significance that points to the data's non-normality.

The table includes, for each normality test, its value, the number of degrees of freedom (df), and the significance of the test for each variable. Consider the examples of the variables NDECode and NDSClass: the Kolmogorov-Smirnov test values are *0,182* and *0,140*, at a significance of *0.001* and *0,033*, respectively. For the same variables, the Shapiro-Wilk test values are *0,821* and *0,923*, with a significance of *0,000* and *0,007*, respectively. With both variables, for both tests, the significance is below *0,05*. Therefore, we cannot assume that our dependent variables come from a population with normal distribution. An analysis of table 6.3 shows that several of our independent variables cannot be assumed to come from a population with a normal distribution either. Others, such as *B_DT_CWAG*, can be assumed to have a normal distribution.

In summary, the pre-conditions for the using parametric tests in our subsequent analysis are not met. Therefore, we will proceed using non-parametric tests.

6.5.2 Data set reduction

Outlier and extreme values can change our view on the relations between dependent and independent variables. For each dependent variable, we conducted a linear regression analysis using the average, best and worst cases of the independent variables. We repeated this analysis for each of our hypotheses and flagged the outliers. This resulted in the removal of four cases in our analysis, with each of the dependent variables. In particular, the reviews of the components produced by the development teams *15*, *19*, and *38* were flagged for DT, PT, RT, and $Diff\_DT\_PT$, while the review of the component produced by development team *25* was flagged for PT, RT and $Diff\_DT\_PT$, but not for DT. The outlier removal is illustrated in figures 6.9(a) and 6.9(b), where cases *15*, *19*, *25*, and *38* are removed from the latter. In both cases, the horizontal scale represents the expertise of the peer team. In this case, expertise is measured by the average grade of the best student in the peer team. The vertical scale represents the number of different bug codes found in the inspected code in the inspection in which that peer

[4]For instance, in [Maroco 03], the Shapiro-Wilk test is recommended for samples with less than *30* cases. However, the same author notes that the manuals for the statistics package we used in this dissertation **SPSS**, `http://www.spss.com/`, recommend using this test for samples with less than *50* cases.

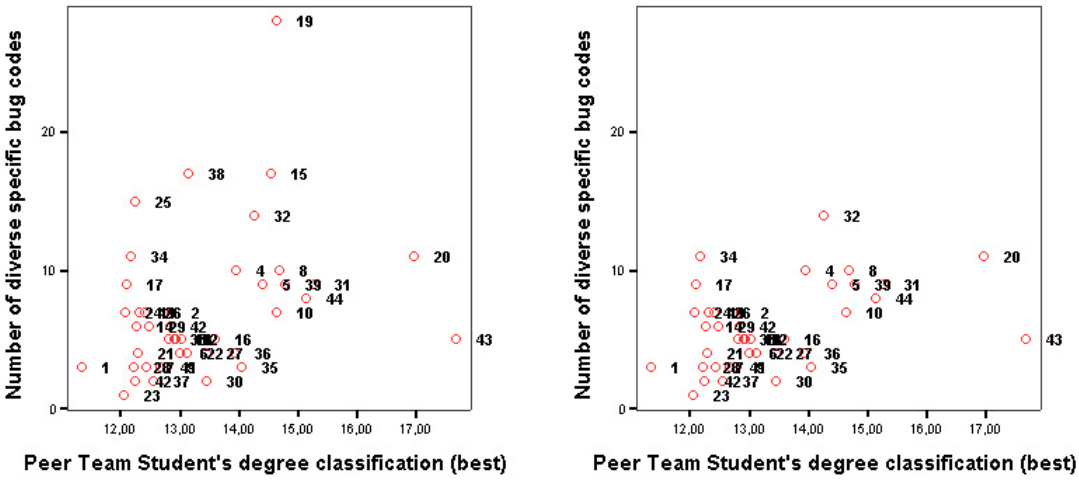
	Kolmogorov-Smirnov(a)			Shapiro-Wilk		
	Statistic	df	Sig.	Statistic	df	Sig.
NDSCode	,182	43	***,001***	,821	43	***,000***
NDGClass	,140	43	**,033**	,923	43	***,007***
A_DT_AG	,188	43	***,001***	,914	43	***,003***
W_DT_AG	,200	43	***,000***	,915	43	***,004***
B_DT_AG	,158	43	***,009***	,894	43	***,001***
A_DT_SWAG	,144	43	**,025**	,948	43	,051
W_DT_SWAG	,166	43	***,005***	,937	43	**,020**
B_DT_SWAG	,141	43	**,031**	,947	43	**,048**
A_DT_CWAG	,133	43	,053	,957	43	,108
W_DT_CWAG	,146	43	**,021**	,953	43	,075
B_DT_CWAG	,111	43	,200(*)	,959	43	,128
A_PT_AG	,182	43	***,001***	,915	43	***,004***
W_PT_AG	,180	43	***,001***	,928	43	**,010**
B_PT_AG	,169	43	***,004***	,884	43	***,000***
A_PT_SWAG	,134	43	,052	,954	43	,084
W_PT_SWAG	,151	43	**,015**	,944	43	**,036**
B_PT_SWAG	,142	43	**,029**	,953	43	,074
A_PT_CWAG	,121	43	,117	,962	43	,162
W_PT_CWAG	,131	43	,061	,955	43	,088
B_PT_CWAG	,111	43	,200(*)	,963	43	,183
A_RT_AG	,081	43	,200(*)	,969	43	,292
W_RT_AG	,199	43	***,000***	,925	43	***,008***
B_RT_AG	,153	43	**,013**	,911	43	***,003***
A_RT_SWAG	,065	43	,200(*)	,988	43	,916
W_RT_SWAG	,175	43	***,002***	,920	43	***,005***
B_RT_SWAG	,151	43	**,015**	,945	43	**,040**
A_RT_CWAG	,069	43	,200(*)	,991	43	,978
W_RT_CWAG	,153	43	**,013**	,930	43	**,011**
B_RT_CWAG	,126	43	,085	,971	43	,337
A_RT_Diff_AG	,096	43	,200(*)	,978	43	,569
W_RT_Diff_AG	,120	43	,132	,974	43	,430
B_RT_Diff_AG	,078	43	,200(*)	,989	43	,953
A_RT_Diff_SWAG	,096	43	,200(*)	,974	43	,426
W_RT_Diff_SWAG	,078	43	,200(*)	,981	43	,674
B_RT_Diff_SWAG	,088	43	,200(*)	,974	43	,418
A_RT_Diff_CWAG	,136	43	**,046**	,968	43	,264
W_RT_Diff_CWAG	,098	43	,200(*)	,975	43	,468
B_RT_Diff_CWAG	,089	43	,200(*)	,971	43	,329

Table 6.3: Normality tests for the independent and dependent variables. The values marked with (*) are lower bounds for the true significance of the Kolmogorov-Smirnov test. (a) stands for Lilliefors significance correction. We cannot assume a normal distribution of the variables in **bold**. The significance of tests is highlighted in **bold** for tests with $p < 0,05$ and ***italic bold*** for tests with $p < 0,01$

.

team participated.

A similar outlier and extreme values detection and removal was carried out for the error classes. We found that, again, the components developed by development teams *15*, *19*, and *38* were flagged as outliers. The component developed by development team *23* was also flagged with about *1/3* of the independent variables. As such, for analysis concerning the error classes variability, we removed from the sample the reviews concerning the components built by these four development teams.



(a) Expertise vs. NDSCode, with outliers (b) Expertise vs. NDSCode, without outliers

Figure 6.9: Number of diverse specific defect codes, by peer team expertise

In summary, for each of the independent values we excluded four cases from our analysis. Note that three of these cases were excluded from both analysis.

6.5.3 Hypothesis testing

We started by performing a correlation analysis, using the both Kendall's and Spearman's correlation tests. Both tests are non-parametric and suited for correlation analysis of our variables. For each of the hypotheses under scrutiny, we tested the correlations of our dependent variables with respect to the independent variables. Note that, due to the data set reduction discussed in the previous section, we only used the *39* cases for each of the correlation analysis.

In both correlation tests, the significance level presented here is the two-tailed asymptotic significance. A significance level below *0,05* indicates a statistically significant correlation, for which there is a probability of less than *5%* that the apparent correlation is only a coincidence. To facilitate the correlation tables analysis, we will highlight such correlations in **bold font**. Correlations with a probability inferior to *1%* of being coincidencidental will be highlighted using the ***bold italic font***. The strength and sign of the correlations will also be presented here. Correlations with a strength close to *1* (in absolute value) are considered very strong, while correlations with a strength closer to *0* are considered weaker. The sign of the correlation indicates whether the variables are expected to have a positive correlation (where an increase in one variable corresponds to an increase on the other), or a negative one (where an increase in one variable corresponds to a decrease in the other variable, and vice-versa).

For the hypotheses where the correlations were shown to exist and be of statistical significance, we further explored the observed relationships. As we were trying to

assess the effect of expertise in the diversity of defects found during code inspections, we divided our sample into quartiles, using the expertise metric in each hypothesis as our criterion for assigning cases to each quartile. For each hypothesis, we ended up with *4* groups of cases. We then tested whether or not these groups came from the same underlying population distribution. The rationale is that, if they do not come from the same underlying population distribution, there are significant differences among those groups. As the groups were formed using the relative expertise of teams as the discriminant factor, we can use those differences to better understand the effect of expertise in the outcome of the code inspections.

H1

Hypothesis **H1** is concerned whether or not developer skill has an effect on the inspected defect diversity. We assessed the correlation of each of our three candidate predictors *AG*, *SWAG*, and *CWAG*, using, for each of them, the mean, best and worst values found in the development team (*DT*). The correlation analysis for **H1** is summarized in table 6.4.

		DT								
		AG			SWAG			CWAG		
Kendall's tau_b		mean	worst	best	mean	worst	best	mean	worst	best
NDECode	corr.	-0,134	-0,096	-0,145	-0,155	-0,164	-0,158	-0,161	-0,178	-0,152
	sig.	0,247	0,407	0,209	0,180	0,157	0,172	0,165	0,124	0,188
	N	39	39	39	39	39	39	39	39	39
NDGClass	corr.	0,012	0,063	-0,014	-0,068	-0,070	-0,036	-0,056	-0,070	-0,030
	sig.	0,922	0,590	0,902	0,565	0,548	0,759	0,633	0,548	0,797
	N	39	39	39	39	39	39	39	39	39
Spearman's rho		mean	worst	best	mean	worst	best	mean	worst	best
NDEClass	corr.	-0,191	-0,142	-0,201	-0,219	-0,204	-0,216	-0,216	-0,223	-0,217
	sig.	0,245	0,388	0,219	0,180	0,213	0,186	0,186	0,172	0,185
	N	39	39	39	39	39	39	39	39	39
NDGClass	corr.	0,011	0,082	-0,024	-0,116	-0,105	-0,072	-0,094	-0,096	-0,067
	sig.	0,947	0,621	0,884	0,481	0,525	0,664	0,569	0,560	0,687
	N	39	39	39	39	39	39	39	39	39

Table 6.4: Correlation analysis for the variables of **H1**.

None of our candidate dependent variables was shown to be correlated with neither of the independent variables. We cannot reject $H1_0$. We can conclude that developer skill had no observable effect on the inspected defect diversity.

H2

Hypothesis **H2** is concerned whether or not the peer team skill has an effect on the inspected defect diversity. We assessed the correlation of each of our three candidate predictors *AG*, *SWAG*, and *CWAG*, using, for each of them, the mean, best and worst values found in the peer team (*PT*). The correlation analysis for **H2** is summarized in table 6.5.

		PT								
		AG			SWAG			CWAG		
Kendall's tau_b		mean	worst	best	mean	worst	best	mean	worst	best
NDECode	corr.	0,254	0,225	0,248	0,241	0,241	0,247	0,241	0,255	0,244
	sig.	**0,028**	0,053	**0,032**	**0,037**	**0,037**	**0,033**	**0,037**	**0,027**	**0,035**
	N	39	39	39	39	39	39	39	39	39
NDGClass	corr.	0,314	0,324	0,265	0,321	0,344	0,289	0,332	0,372	0,280
	sig.	***0,008***	***0,006***	**0,024**	***0,006***	***0,003***	**0,014**	***0,005***	***0,002***	**0,017**
	N	39	39	39	39	39	39	39	39	39
Spearman's rho		mean	worst	best	mean	worst	best	mean	worst	best
NDECode	corr.	0,361	0,315	0,379	0,354	0,344	0,363	0,357	0,364	0,361
	sig.	**0,024**	0,051	**0,017**	**0,027**	**0,032**	**0,023**	**0,025**	**0,023**	**0,024**
	N	39	39	39	39	39	39	39	39	39
NDGClass	corr.	0,413	0,429	0,360	0,440	0,476	0,402	0,459	0,505	0,399
	sig.	***0,009***	***0,006***	**0,024**	***0,005***	***0,002***	**0,011**	***0,003***	***0,001***	**0,012**
	N	39	39	39	39	39	39	39	39	39

Table 6.5: Correlation analysis for the variables of **H2**.

From Table 6.5 we can observe significant correlations between our independent and dependent variables for hypothesis **H2**. The only exception is the correlation between *NDSCode* and *W_PT_AG*. Note that even these correlations have a significance very close to *0,05* (*0,053* and *0,051*, for Kendall's and Spearman's correlations, respectively). In general, the correlations are stronger and more significant with *NDGClass* than with *NDSCode*. From all these correlations, the strongest and most significant is between *NDGClass* and *W_PT_CWAG*. The expertise of the worst member of the peer team has consistently the highest correlation with *NDGClass*, regardless of the particular expertise metric, for the different number of generic defect classes. There is no clear superiority of any of the expertise metrics, with respect to their correlation with the number of specific defect codes.

We divide our sample into quartiles, using each of the expertise metrics in turn, in order to obtain, for each of the metrics, four groups of approximately the same cardinality, where quartile *1* includes the peer teams with the lowest expertise, and so on, until the 4^{th} quartile, which includes the peer teams with the highest expertise. The quartile including a given peer team may vary according to the specific expertise metric used.

Intuitively, we want to check whether or not there are significant differences between the members of each of the quartiles, with respect to the diversity of errors found. As the diversity of errors found does not follow a normal distribution, we will have to use a non-parametric test. The Kruskal-Wallis test [Kruskal 52] is adequate for our purposes. It is the non-parametric alternative to using the one-way ANOVA, when ANOVA's pre-conditions (normality and homocedasticity) are not met. The Kruskal-Wallis test is performed on ranked data. The measurement observations (*NDSCode*) are converted to their ranks in the overall data set: the smallest value gets a rank of 1, the next smallest gets a rank of 2, and so on. The null hypothesis for this test states that the mean ranks of the samples (quartiles) are expected to be the same. The alternative

is that at least one of them is not.

If θ_i is the mean rank in quartile i, our null hypothesis is:

$H_0 : \theta_1 = \theta_2 = \theta_3 = \theta_4$

The alternative hypothesis, H_1 is:

$H_1 : \exists i, j : \theta_i \neq \theta_j (i \neq j; i, j = 1, ..., 4)$

Table 6.6 summarizes the test statistics of the Kruskal-Wallis test for the diversity of defect codes, using the peer team's expertise quartile as grouping variable. With 3 degrees of freedom (*number of quartiles* $-$ 1), the significance of this test is $p < \alpha = 0,100$, with any of our candidate expertise measurements. In other words, we reject the null hypothesis, as at least one of the quartiles presents significant differences, when compared to the others.

	PT								
	AG			SWAG			CWAG		
	mean	worst	best	mean	worst	best	mean	worst	best
Chi-Square	6,412	8,589	9,111	6,767	8,715	11,077	10,234	12,021	11,077
df	3	3	3	3	3	3	3	3	3
Asymp. Sig.	**0,093**	**0,035**	**0,028**	**0,080**	**0,033**	***0,011***	***0,017***	***0,007***	***0,011***

Table 6.6: Kruskal-Wallis test for hypothesis **H2**, using NDDCode

The mean rank number of different defect codes found on the inspections follows the same general pattern, regardless of the specific metric for assessing expertise: the inspections with the peer team members of the forth quartile (the peers with the highest expertise) have, consistently, the highest mean of ranks. These differences are statistically significant, according to the results of the Kruskal-Wallis test.

To facilitate the interpretation of this test, consider the boxplots in figure This trend is illustrated in figure 6.10.

When there is a natural *a priori* ordering for the different populations (as it happens with the quartiles), the Jonckheere-Terpstra (**J-T**) test is more powerful than the Kruskal-Wallis[5]. In this test, the null hypothesis that the members of the different quartiles will have the same defect code diversity is tested against the alternative that as the quartile increases the defect code diversity will change. The results of the J-T test are summarized in table 6.7, where we present the **number of levels**(*i.e.* number of different groups), **number of cases** tested, the **observed J-T statistic**, the **mean J-T statistic**, the **standard deviation of the J-T statistic**, the **standardized** (normalized) **J-T statistic**, and the **asymptotic two-tailed significance** of the test.

All the J-T tests are significant with $p < \alpha = 0,05$, and the tests using the expertise of the worst member of the peer team were significant with $p < \alpha = 0,01$. Note the normalized J-T statistic has a positive value with all metrics. In this experiment, a positive value of the normalized J-T statistic denotes an increase of *NDSCode*, from

[5]See, for instance, `http://www.morris.umn.edu/~sungurea/introstat/nonparametric/learningtools.html`, or the SPSS 14 statistics guide

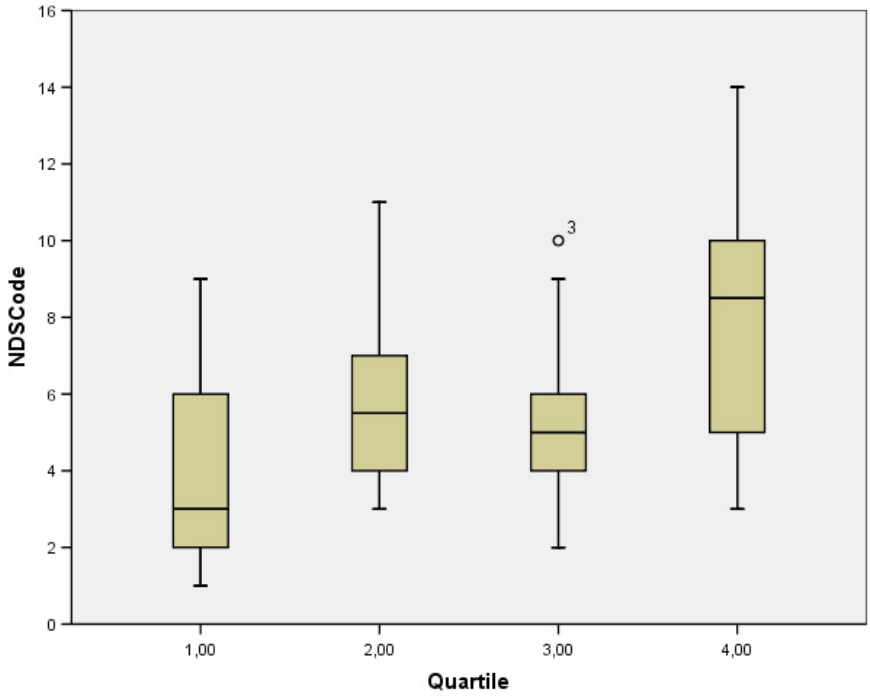


Figure 6.10: Reported NDScode distribution, grouped by W_PT_CWAG quartiles

	PT								
	AG			SWAG			CWAG		
	mean	worst	best	mean	worst	best	mean	worst	best
N. Levels	4	4	4	4	4	4	4	4	4
N	39	39	39	39	39	39	39	39	39
Obs. J-T	383,500	394,000	368,000	375,000	396,000	371,000	379,000	408,000	371,000
Mean J-T	285,000	285,000	285,000	285,000	284,500	285,000	285,000	285,000	285,000
Stddev J-T	39,609	39,609	39,609	39,609	39,588	39,609	39,609	39,609	39,609
Std. J-T	2,487	2,752	2,095	2,272	2,817	2,171	2,373	3,105	2,171
Sig. (2-t)	**0,013**	***0,006***	**0,036**	**0,023**	***0,005***	**0,030**	**0,018**	***0,002***	**0,030**

Table 6.7: Jonckheere-Terpstra test for the number of specific defect codes.

lower to higher quartiles. In summary, the J-T tests, confirmed the results obtained previously with the Kruskal-Wallis tests.

Overall, although the correlations between the expertise metrics and *NDSCode* are not very strong, the assignment of groups into quartiles and the comparison between those groups shows a trend that favors the usage of peers with a high expertise, as expected.

We will follow a similar process for our metric of the diversity of defects' classes found during inspections, *NDGClass*. Table 6.8 summarizes the Kruskal-Wallis tests results.

Unlike what we have observed with respect to correlations between the independent variables and *NDGClass*, there is no statistically significant difference between the quartiles formed using four (*A_PT_AG*, *W_PT_AG*, *A_PT_SWAG*, and *W_PT_SWAG*) of our nine independent variables. The remaining five variables can be used to divide the sample into quartiles in such a way that at least one of those quartiles is signif-

	PT								
	AG			SWAG			CWAG		
	mean	worst	best	mean	worst	best	mean	worst	best
Chi-Square	5,787	5,262	9,010	4,661	4,671	7,817	7,148	8,775	7,393
df	3	3	3	3	3	3	3	3	3
Asymp. Sig.	0,122	0,154	**0,029**	0,198	0,198	**0,050**	**0,067**	**0,032**	**0,060**

Table 6.8: Kruskal-Wallis test for defect classes.

icantly different than the others, with respect to the mean *NDGClass*. The J-T test, summarized in table 6.9 does not confirm most of the results of the Kruskal-Wallis test. Although, according to the J-T test, there are also five variables for which the test statistic is significant, the interception between the sets of metrics we would select from each tests has only two metrics: *A_PT_CWAG* and *W_PT_CWAG*.

	PT								
	AG			SWAG			CWAG		
	mean	worst	best	mean	worst	best	mean	worst	best
N. Levels	4	4	4	4	4	4	4	4	4
N	39	39	39	39	39	39	39	39	39
Obs. J-T	352,500	363,500	337,500	358,500	369,500	351,000	365,000	394,000	355,500
Mean J-T	285,000	285,000	285,000	285,000	285,000	285,000	285,000	285,000	285,000
Stddev J-T	39,415	39,415	39,415	39,415	39,415	39,415	39,415	39,415	39,415
Std. J-T	1,713	1,992	1,332	1,865	2,144	1,674	2,030	2,765	1,789
Sig. (2-t)	0,087	**0,046**	0,183	0,062	**0,032**	0,094	**0,042**	***0,006***	0,074

Table 6.9: Jonckheere-Terpstra test for defect classes.

In summary, we can reject the null hypothesis $H2_0$. We were able to find several measures of the expertise within the peer team which can be used as predictors of the diversity of the reported defects. The overall tendency, more visible with *NDSCode*, favors the benefits of using expert peers in the review teams, as expected. When using a more coarse grained metric for the diversity of defects fount, this tendency is not supported by all the statistic tests we conducted in this experiment, although it is partially confirmed by some of them. Nevertheless, two of the expertise metrics did pass all the tests we conducted, so we can consider them good candidates for predictors of *NDGClass*.

H3

Hypothesis **H3** concerns whether or not the review team skill has an effect on the inspected defect diversity. We assessed the correlation of each of our three candidate predictors *AG*, *SWAG*, and *CWAG*, using, for each of them, the mean, best and worst values found in the review team (*RT*). The correlation analysis for **H3** is summarized in table 6.10.

The expected influence of the overall review team skill in the outcome of the inspections, in terms of the diversity of the defect codes and classes, was not confirmed when using the *SWAG* and *CWAG* metrics of expertise, with any of their variants. There is no

		RT								
		AG			SWAG			CWAG		
Kendall's tau_b		mean	worst	best	mean	worst	best	mean	worst	best
NDECode	corr.	0,140	0,149	0,093	0,058	-0,024	0,081	0,063	-0,001	0,073
	sig.	0,227	0,204	0,428	0,617	0,836	0,487	0,583	0,990	0,534
	N	39	39	39	39	39	39	39	39	39
NDGClass	corr.	0,280	0,210	0,240	0,165	0,085	0,200	0,194	0,123	0,197
	sig	**0,017**	0,077	**0,043**	0,159	0,469	0,091	0,098	0,297	0,095
	N	39	39	39	39	39	39	39	39	39
Spearman's rho		mean	worst	best	mean	worst	best	mean	worst	best
NDECode	corr.	0,159	0,202	0,133	0,090	-0,013	0,114	0,100	0,008	0,102
	sig.	0,334	0,218	0,420	0,584	0,938	0,490	0,545	0,963	0,538
	N	39	39	39	39	39	39	39	39	39
NDGClass	corr.	0,367	0,290	0,323	0,227	0,110	0,253	0,255	0,165	0,249
	sig.	**0,021**	0,074	**0,045**	0,164	0,503	0,121	0,117	0,315	0,126
	N	39	39	39	39	39	39	39	39	39

Table 6.10: Correlation analysis for the variables of **H3**.

correlation between their value and the diversity of defects found. There is, however, a correlation between the *AG* of the review team, both when using the mean value and the best value within the team. We will further explore these two correlations. Table 6.11 presents the Kruskal-Wallis test for the impact of each of the expertise metrics on the diversity of detected defects.

	RT_AG	
	mean	best
Chi-Square	4,438	3,658
df	3	3
Asymp. Sig.	0,218	0,301

Table 6.11: Kruskal-Wallis test for **H3** defect classes.

The Kruskal-Wallis tests provide us with no evidence of having at least a group for which the the mean number of detected classes is significantly different from the remaining ones. To confirm these results, we also conducted a J-T test on the same variables. Table 6.12 summarizes the results of the J-T test.

	RT_AG	
	mean	best
N. Levels	4	4
N	39	39
Obs. J-T	362,500	345,500
Mean J-T	285,000	284,500
Stddev J-T	39,415	39,394
Std. J-T	1,966	1,548
Sig. (2-t)	**0,049**	0,122

Table 6.12: Jonkeera-Terpstra test for **H3** defect classes

Unlike the results of the Kruskal-Wallis test, the Jonkeera-Terpstra test has shown a significant trend in the data, when using the *A_RT_AG* metric. According to the results of this test, one can observer a trend of a growing number of reported defects, as the

overall expertise of the review team increases. Figure 6.11 presents the boxplots for the distribution of detected defect classes in the four quartiles.

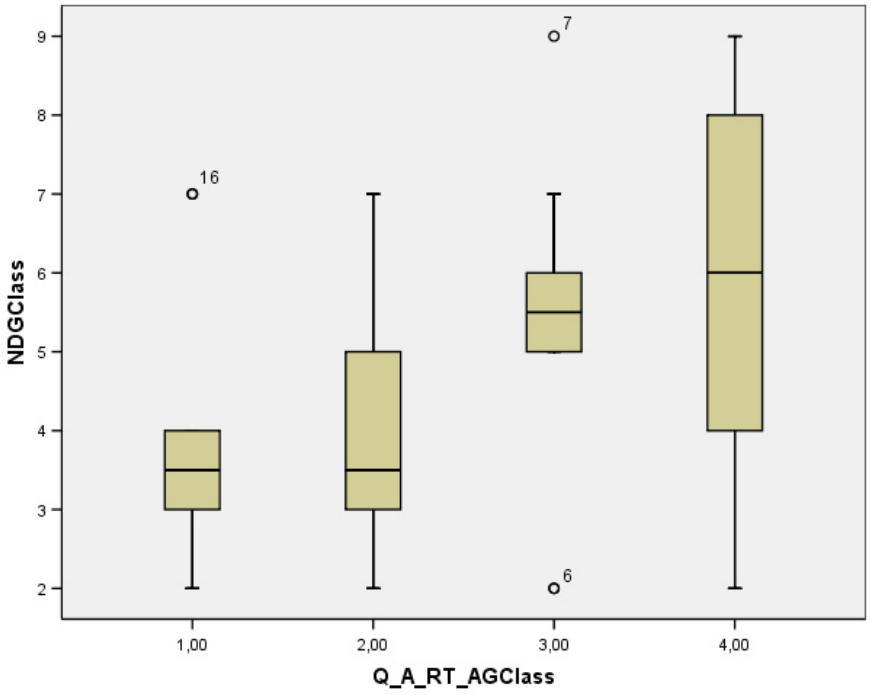


Figure 6.11: Reported NDGClass distribution, grouped by A_RT_AG quartiles

The problem that made these distributions fail in the Kruskal-Wallis test for significant differences relates to the high variance within each of the groups. Although the mean value of detected defect classes grows with the expertise of the review team, the dispersion within each quartile makes the rank-based test loose significance. In conclusion, we were able to find one metric that supports $H3_1$, that is, the review team expertise has an effect on the inspected defect diversity, when considering a coarse grained assessment of defect diversity, although this conclusion does not hold for the fine-grained metric of defect diversity. The dispersion of results in all quartiles suggests that the inner dynamics of the review team should be better assessed. This is the focus of hypothesis **H4**.

H4

Hypothesis **H4** concerns whether or not the difference between the skill of the developers and the skill of the peers has an effect on the inspected defect diversity. We assessed the correlation of each of our three candidate predictors *AG*, *SWAG*, and *CWAG*, using, for each of them, the mean, best and worst values found in the differences within the review team (*RT_Diff*). The correlation analysis for **H4** is summarized in table 6.13.

With the exception of *W_AG_RT_Diff*, all the predictors for **H4** have a significant negative correlation with *NDSCode*. In general, these correlations are stronger with *SWAG* and *CWAG* than with *AG*. We will use the Kruskal-Wallis test to detect signifi-

		RT_Diff								
		AG			SWAG			CWAG		
Kendall's tau_b		mean	worst	best	mean	worst	best	mean	worst	best
NDECode	corr.	-0,243	-0,195	-0,262	-0,292	-0,278	-0,233	-0,306	-0,269	-0,250
	sig.	**0,036**	0,092	**0,023**	**0,012**	**0,016**	**0,044**	**0,008**	**0,020**	**0,031**
	N	39	39	39	39	39	39	39	39	39
NDGClass	corr.	-0,209	-0,155	-0,214	-0,286	-0,303	-0,208	-0,289	-0,280	-0,240
	sig.	0,076	0,186	0,068	**0,015**	**0,010**	0,076	**0,014**	**0,017**	**0,041**
	N	39	39	39	39	39	39	39	39	39
Spearman's rho		mean	worst	best	mean	worst	best	mean	worst	best
NDECode	corr.	-0,353	-0,270	-0,376	-0,416	-0,384	-0,356	-0,444	-0,382	-0,383
	sig.	**0,028**	0,097	**0,018**	**0,008**	**0,016**	**0,026**	**0,005**	**0,017**	**0,016**
	N	39	39	39	39	39	39	39	39	39
NDGClass	corr.	-0,282	-0,216	-0,280	-0,398	-0,422	-0,298	-0,401	-0,397	-0,337
	sig.	0,082	0,187	0,084	**0,012**	**0,007**	0,066	**0,011**	**0,012**	**0,036**
	N	39	39	39	39	39	39	39	39	39

Table 6.13: Correlation analysis for the variables of **H4**.

cantly different groups with respect to the impact of the expertise gap between developers and peers in the detected defect code diversity (table 6.14).

	RT_Diff								
	AG			SWAG			CWAG		
	mean	worst	best	mean	worst	best	mean	worst	best
Chi-Square	5,185	2,455	8,881	11,270	7,924	6,652	10,767	6,941	5,349
df	3	3	3	3	3	3	3	3	3
Asymp. Sig.	0,159	**0,483**	**0,031**	***0,010***	**0,048**	**0,084**	***0,013***	**0,074**	0,148

Table 6.14: Kruskal-Wallis test for **H4** defect codes.

We will confirm these results with the J-T test (table 6.15).

	RT_Diff								
	AG			SWAG			CWAG		
	mean	worst	best	mean	worst	best	mean	worst	best
N. Levels	4	4	4	4	4	4	4	4	4
N	39	39	39	39	39	39	39	39	39
Obs. J-T	226,000	225,000	211,500	177,500	191,000	197,500	171,000	181,000	202,000
Mean J-T	285,000	285,000	285,000	285,000	285,000	285,000	285,000	285,000	285,000
Stddev J-T	39,609	39,609	39,609	39,609	39,609	39,609	39,609	39,609	39,609
Std. J-T	-1,490	-1,515	-1,856	-2,714	-2,373	-2,209	-2,878	-2,626	-2,095
Sig. (2-t)	0,136	0,130	0,064	***0,007***	**0,018**	**0,027**	***0,004***	***0,009***	**0,036**

Table 6.15: Jonckheere-Terpstra Test for **H4** defect codes.

When using the coarse grained metric for defect diversity, *NDGClass*, only five of the expertise metrics are significantly correlated to it. None of the *AG* metrics is correlated to *NDGClass*. As with the previous hypotheses, we will use the Kruskal-Wallis test to check whether there are significantly different results between the quartile-based groups. Table 6.16 summarizes the results for these tests.

Three out of the five metrics which were negatively correlated to *NDGClass* lead to groups with significant differences in their mean *NDGClass*. Table 6.17 summarizes the J-T test results, used to confirm our findings.

	RT_Diff								
	AG			SWAG			CWAG		
	mean	worst	best	mean	worst	best	mean	worst	best
Chi-Square	2,319	1,076	4,993	8,522	7,854	3,549	11,514	7,589	7,149
df	3	3	3	3	3	3	3	3	3
Asymp. Sig.	0,509	0,783	0,172	**0,036**	**0,049**	0,314	***0,009***	**0,055**	**0,067**

Table 6.16: Kruskal-Wallis test for **H4** defect classes.

	RT_Diff								
	AG			SWAG			CWAG		
	mean	worst	best	mean	worst	best	mean	worst	best
N. Levels	4	4	4	4	4	4	4	4	4
N	39	39	39	39	39	39	39	39	39
Obs. J-T	241,000	242,000	216,000	189,000	183,500	210,000	174,500	184,500	199,500
Mean J-T	285,000	284,000	285,000	285,000	285,000	285,000	285,000	285,000	284,000
Stddev J-T	39,415	39,374	39,415	39,415	39,415	39,415	39,415	39,415	39,374
Std. J-T	-1,116	-1,067	-1,751	-2,436	-2,575	-1,903	-2,803	-2,550	-2,146
Sig. (2-t)	0,264	0,286	0,080	**0,015**	**0,010**	0,057	***0,005***	**0,011**	**0,032**

Table 6.17: Jonckheere-Terpstra Test for **H4** defect classes.

In the J-T test, the quartiles based on five metrics, including the three we highlighted when discussing the Kruskal-Wallis test, had significantly different means of *NDGClass*.

The average number of different reported bug codes and classes decreased, when comparing the first with the last quartiles. In other words, the number of diverse defect codes and classes decreases as we move from development teams with lower expertise than their peer teams to the opposite case. As such, we can reject the null hypothesis $H4_0$. Several metrics of the difference between the expertise of the members of the development and peer teams can be used as predictors of the diversity of the reported defects.

6.6 Interpretation

6.6.1 Evaluation of results and implications

H1

We expected the best developers to produce components with less variety of defects, but this was not confirmed by our experiment. Assuming that, in general, the best students produce code of a higher quality, there are a number of possible alternative explanations for these results. Our experimental design does not use any information concerning neither the relative severity of the defects found in inspections, nor their expected impact on maintenance. Moreover, we used defect code and class diversity, but not the actual number of defects found during the code inspections, so it may be the case that the quality difference manifests itself through the number of defects. Last, but not the least, although all participants were implemeting components with the same

interface specification, the internal sophistication of the implementation of the components could vary. If the best developers chose more sophisticated implementations and sophisticated implementations are more prone to defects than simpler ones (due to their complexity), then the added complexity could cancel out the effect of the expertise of developers. In short, it may be the case that this assessment of the impact of expertise of developers in the software artifact's quality is too simplistic.

It may also be the case that, because developer teams were also part of the review teams, their expertise countered the effect of a lower variety of problems with that of a higher efficiency in finding them.

H2

As expected, we observed that the expertise of the peer team does have a positive effect on the variety of problems uncovered during code inspections. This effect is more noticeable (and statistically significant) when comparing the performance of the review teams where the peer teams had the best experts. This suggests an effect of leadership of expert peers in the inspection teams. The contrast of expertise is also more noticeable when comparing the 4^{th} quartile with the previous ones. This partially results from the skewed distribution of expertise, discussed earlier in sub-section 6.5.1.

H3

The expertise of the whole review team did not show a significant relationship with the outcome of the review. The considerations concerning a possible oversimplification of our dependent variable, combined with the cancellation effect also described with respect to **H1** may be responsible for this discrepancy between the expected result and the outcome of this experiment. Nevertheless, it is worth noticing the relationship observed between two of the expertise metrics of the review team (*A_RT_AG* and *B_RT_AG*) and the diversity of defects found in the inspections. Although not very strong this relationship was statistically significant.

H4

As expected, when peer teams of low expertise analyze the work of development teams with a higher expertise, the outcome of the code review shows a lower variety of defects found. Conversely, more defects are found in inspections where the peer teams have a higher expertise than the one of the development teams. A potential leadership effect of a reviewer over the others is not visible from the data analyzed while testing this hypothesis.

With the experiment design of this last hypothesis, we have an alternative perspective on the inspection group dynamics, when compared to hypothesis **H3**. On **H3** we had no indication of how the expertise was distributed within the group, thus being

vulnerable to the cancellation effect occurring when (i) having good experts examining their own code and not finding many problems with it, because they were not there, or (ii) weaker programmers examining their own code and not realizing the problems in it.

Both situations lead to a cancellation effect that might explain the unexpected results with hypothesis **H3**.

There is a curious effect in the evolution of the variety of defects found between the second and third quartiles of **H4** (the second quartile has development teams with a lower expertise than their peer teams, while the third inverts this relationship). One could expect the variety of defects to be lower on the third quartile, following the tendency found from the first to the forth quartiles. However, the expertise level is very close, within groups 2 and 3. Therefore, it may be the case that it is the domain level expertise that dominates the outcome of the inspection. With a better knowledge of the deliverables being inspected, allied with a slightly better expertise than their peers, the authors may be responsible for this locally increased benefit of the code review. As the gap of expertise between development team and peer team members widens, this effect would be mitigated by the dominating effect of the higher code quality and lower external reviewer expertise.

6.6.2 Threats to validity

In what concerns the **internal validity of the study**, we consider two sorts of validity threats: social threats and single group threats. **Social threats** to internal validity could stem from the usage of differentiated treatments within our sample. As this was not the case, we can dismiss their potential effect. **Single group threats** can occur from not using a control group with this experiment. The potential threats to validity of this experiment include:

- **History**. To the best of our knowledge, this potential threat had no effect in our experiment. Each code artifact was only inspected once, and all inspections took place in a very limited time span (two consecutive days).

- **Maturation**. Occurs when subjects react differently as time progresses, for instance, as a result of a learning process. Concerning the development of the components that were inspected, this risk was mitigated by the usage of a programming language well-known by all subjects. With respect to the subjects' performances in reviews, all subjects were inexperienced in the used technique and participated in two consecutive reviews (inspections took place in two hours sessions, with one hour dedicated to each of the inspections where subjects participated). The potential learning effect between both reviews is mitigated both by the lack of time between the reviews and by the change of roles between reviews. Note that no inspection team was simultaneously in charge of two dif-

ferent inspections. In other words, if development team A's work was inspected with the help of peer team B, then when team B acted as developer team B, their work was inspected by peer team C, whose elements were not the same as the ones of team A. So, no maturation effect resulting from a repetition of inspection teams, even if with a switch of roles was allowed by this design.

- **Testing**. There was no repetition of activities during the experiment, thus avoiding this validity threat.

- **Instrumentation**. All participants used the same defect reporting template, so problems with instrumentation could only occur from mis usage of the template. We observed that participants did not consistently register defects, with respect to their frequency on the artifact. Consider the following example: suppose the development team did not follow the naming conventions when declaring a variable; while in some of the reviews this problem was reported only once, with the comment "repeated throughout the code in all occurrences of variable *variableIdentifier*", others reported each and every occurrence of this problem in the source as a separate defect. So, if the variable was used ten times, the problem would be reported ten times, as well, in the latter case, or just once, in the former. As a result, the number of defects found was not collected consistently. We used the number of different defect classes, rather than the total number of defects found, thus avoiding this data collection problem.

- **Statistical regression**. This threat is not applicable to our experimental design.

- **Selection**. No selection was performed on the subjects of this experiment. All students enrolled in the course who actively participated in the projects were used as subjects, so the subjects are representative of the population.

- **Mortality**. 6 out of the 93 students enrolled in the course were excluded. 5 of these were *"ghost students"* - although they were enrolled in the course, they never showed up on a single class. The other subject dropped out the course during the requirements specification phase, prior to the development and review phases under evaluation in this experiment. With only one student dropping off an optional course, we believe there is no cause for concern with mortality in this experiment.

- **Ambiguity about direction of causal influence**. The assessment of subject's expertise was independent from the assessment of their performance in the experiment and based on information collected **before** the beginning of the experiment. If any, causal relationships could only be established from expertise to performance, and not otherwise.

Multiple groups threats do not apply in this experiment. All subjects were under the same conditions, in what concerns their participation in this experiment.

External validity refers to our ability of generalizing results beyond the scope of this experiment. We consider three potential sources of threats:

- **People**. In this experiment, all subjects were informatics students at Universidade Nova de Lisboa. The subjects were not professional developers. This limits our ability to generalize the results of the experiment. Based on the work of Höst *et al.* [Höst 00], already discussed in section 3.3.2, we have reasons to think students can be used as surrogates for novice professional developers, in the context of this experiment, without significantly changing its outcome. Another potential threat may be a lack of diversity of competences among subjects, in the code reviews. Typically, reviewers would include people with different tasks in the software process (e.g. programmers, system engineers, and testers). This shortcoming is somewhat mitigated by the informatics course large spectrum, on the one hand, and the design of the inspection meetings, on the other. To introduce diversity of roles, students were always assigned as reviewers of two different projects. In one of them, their perspective would be that of the authors of the artifact. In the other one, their perspective would be the perspective of a user of the inspected artifact.

- **Setting**. This kind of experiment can be jeopardized by using an obsolete experimental environment. In this case, subjects used modern Java development environments, during development. Concerning the code inspection, although no specific code inspection tool was used, subjects were able to record the problems found during inspection in a special spreadsheet built for this purpose. Violations to some of the inspection guidelines described in the problem checklist could be automatically detected with automated code inspection tools, but these were not used. Another potential issue is the usage of toy examples in these experiments, which may lead to conclusions that do not scale up. The components developed in this course were comparable to fine-grained components. In what concerns inspections, this was not an issue because inspections focus usually on a limited portion of the software. In this case, we used the source code of one of those fine-grained components as the inspection target.

- **Time**. We are not aware of any special events that might have conditioned the component development and code inspection activities. They were performed during the normal semester (roughly one month before the end of classes), with no noticeable special conditions in the student's workload.

With respect to construct validity threats, we consider two categories: social and design threats. Typical social threats include:

- **Hypothesis guessing**. The inspections were carried out as part of a wider development project. Although the students were aware of our intention to use data collected during the project for experimental purposes, they did not know which parts of the project would be used in the experiment. As far as they were concerned, the inspection was just another task they had to perform. This mitigates the potential hypothesis guessing effect that could have occurred if they were aware of the objectives of the experiment.

- **Evaluation apprehension**. Informatics students are very used to development projects as part of their courses. The inspection reports were one out of about ten deliverables they had to submit for their course evaluation, in this experiment, so the specific weight of this task in their final evaluation was fairly small.

- **Experimenter's expectancies**. We did expect to observe significant differences in the results obtained by different combinations of skills within the inspection teams, the training and directions provided to all participants were similar. That said, the selection of inspection teams was blind with respect to the objectives of the experiment, and no intervention on the conduction of the code reviews was made, so that we would not interfere with the inspections outcome.

Construct validity design threats result include:

- **Inadequate pre-operational explication of constructs**. Terms like "practitioner's expertise" are frequently used, not necessarily following always the same definition. Our metrics definition approach mitigates this problem by supporting a rigorous definition of all the metrics used in this experiment.

- **Mono-operation bias**. This threat is mitigated by the fact that our experiment involves over 40 different inspection teams, with all possible combinations of skills filling the different roles in the inspection meetings.

- **Mono-method bias**. Our experimental setting only mitigates this threat partially. We use three different measures for assessing the expertise of team members, and three different ways of assessing the general expertise of each team, to minimize potential biases resulting from our expertise assessment. Nevertheless, they are all based in the average grade of the subjects and this can be argued not to be a perfect expertise assessment. We use two different measures of detected defects diversity. Again, these are variations of the same concept.

- **Confounding constructs and level of constructs**. Ideally, a differentiating replication of this experiment could use another measure of defect detection success, such as the percentage of detected errors. For practical reasons, using this complementary measure of inspection success was not an option in our experimental context. We had no way of measuring the extent to which the inspections were

successful by having an independent detection of defects process that would be, at the same time, complete. So, the lack of an account for the defects not detected in inspections is a potential confounding factor for the interpretation of this experiment's results.

- **Interaction of different treatments**. As a single treatment is involved in our experiment, this threat is dealt with by our experimental design.
- **Interaction of testing and treatment**. This threat was minimized as much as possible, by the fact that subjects were not aware of the fact that the experiment would focus on this particular part of the software development process they were conducting.
- **Restricted generalizability across constructs**. To the best of our knowledge, there were no undesired side-effects as an indirect outcome of this experiment that could impact the remainder of the development process.

The potential threats to conclusion validity include:

- **Statistical power** Although our sample has more cases than the typical controlled experiment in Software Engineering and our sample size was not, *per se*, an obstacle to the usage of parametric tests, several of the variables used in this experiment did not have a normal distribution in the collected data. This forced the usage of non-parametric tests that, in general, have less statistic power than parametric tests. Our confidence in the results increases through the usage of alternative tests both for the correlation analysis and the comparison of the quartile groups for detecting differences between them.
- **Violated assumptions of statistical tests**. To the best of our knowledge, all the assumptions of the used statistical methods were met.
- **Fishing and the error rate**. We tested a fairly small number of hypotheses, and the results obtained in our tests were consistent, in general, regardless of the particular variation we were using in them, so we have no reason to believe the relations found between variables are spurious.
- **Reliability of measures**. As mentioned earlier, we chose measures for which we had a high confidence, with respect to their reliability, to mitigate this threat. We can offer two examples of measures that would bring up this threat, if used in our analysis: the total number of defects found in the inspection, and the individual preparation time recorded by participants. We noted some instability with respect to the first one, and had no way of verifying the accuracy of the second one, particularly because it was too vulnerable to evaluation apprehension by the subjects (participants who did not prepare for the meeting adequately could easily lie about it, for instance).

- **Reliability of treatment implementation**. All subjects had, to the best of our knowledge, similar conditions for performing their inspection roles. The artifacts were distributed to participants with a similar time advance so that they could do their solo reviews, inspections took place during classes and took roughly one our to complete, each.

- **Random irrelevances in experimental setting**. We did not identify any potential sources of variation exogenous to our tests that might have had an influence in their result, nevertheless.

- **Random heterogeneity of subjects**. This threat is mitigated by the number of subjects participating in the experiment and the objectives of the experiment itself. While heterogeneity among subjects certainly existed, it was partly accounted for by the expertise assessment.

6.6.3 Inferences

The evidence collected throughout this experiment suggests that expertise assessments can be used to guide the selection of review team members, to improve the outcome of code reviews. The academic records of participants were used in this experiment to provide the expertise assessment. The results obtained by our participants can be extrapolated for graduate students from the informatics degree of Universidade Nova de Lisboa. Assuming that the students of Universidade Nova de Lisboa are essentially similar to those of other universities with a degree of a similar profile, the results could also be extrapolated for those students, as well. This assumption should be tested, of course, by replicating this experiment in such universities.

Evidence collected elsewhere [Runeson 03] also suggests that the results obtained by students of a profile similar to our participants are close to those obtained by novice professionals, so we can expect these results to hold in that population, as well. Again, this inference could be supported by conducting replications in professional environments.

As practitioners gain experience in a professional environment, the usefulness of the expertise metrics discussed in this experiment is less likely to hold. So, in a professional environment, an alternative expertise assessment should be sought, in order to reflect the performance of code reviewers in their professional context, rather than in an academic one. An alternative and more accurate metric could be an expertise assessment based on the performance of practitioners in previous code reviews. This option was successfully used in [Biffl 02], as we have discussed in the related work section of this chapter.

Last, but not the least, all these inferences assume a similar inspection model is in place. Changes to the inspection model should reduce the probability of obtaining similar results. In particular, introducing a more sophisticated reading technique designed

provide repeatability to the defect detection would render this approach useless. Reading by stepwise abstraction [Dyer 92a], and perspective-based reading [Basili 96b] are examples of reading techniques that, in principle, should ensure that the results of inspections are independent from the practitioners performing them.

6.6.4 Lessons learned

While conducting the experiment, we noted some process issues that could be improved, in future replications of this experiment. We would like to highlight the following:

- A clearer description of the defects collection procedure should be used, with respect to the approach that should be followed when the same defect is found in the software several times. A typical example of a repeated defect would be using an identifier for a variable that does not follow the stipulated naming convention. Without clear guidelines stating whether this defect should be reported and located only once, in the variable's declaration, or in all its occurrences, some reviewers will go for the first option, while others will follow the latter. It follows that such discrepancies are sufficient to hamper our ability to use the total number of defects found in a code artifact in a sound manner.

- Although we collected information concerning the preparation time before the inspection meeting by each inspector, we found the collected information to be unreliable and chose not to use that information in our data analysis. Rather than an exact account of the time spent in the defects detection phase, we noted that a roughly approximated figure was a lot more common. The mode value reported was *1* hour, which was also the recommended preparation time, during the inspection training. With very few exceptions, the reported times were multiples of *30* minutes which leads us not to trust the accuracy of this data. An alternative would be to conduct this preparation phase in a controlled environment, rather than leaving it for participants to do it on their spare time. However, as this controlled environment would probably be a class environment, the duration of the class would then be an external bias to the time spent in the individual defect detection phase.

6.7 Conclusions and future work

6.7.1 Summary

We described an experiment carried out to help understanding the effect of practitioner's expertise in the deliverables produced in the context of CBD.

We focused our attention on the outcome of code inspections, and, in particular, on the variety of problems reported during those inspections. We confirmed the expected positive effect of the expertise of the peer review teams in the outcome of the inspections, observable through the increased variety of defects found when peer experts were available. We also confirmed that having expert peers collaborating in the inspection of components developed by less skilled peers has a positive impact on the outcome of the review.

Moreover, there is also a learning effect, not studied here but vastly commented on the literature, when combining experts with non-experts. This is also expected for the opposite case, where non-experts participate on the review of code developed by experts. However, in this case, a lower variety of defects is found, both because the code is likely to have a higher quality, and because the external reviewers have less capacity to detect its problems. Given the main goal of inspections (maximizing defect detection), the results are poorer.

When observed in isolation, the expertise of the development teams did not show a significant relationship with the variety of problems found. The expertise of the review teams, as a whole, was also not shown to be a good indicator of the outcome of the inspection. Further research is required to determine whether these were the results of cancellation effects of expertise, or if more sophisticated review outcome metrics should have been used here.

6.7.2 Impact

A possible impact of this research is that by carefully selecting the inspection team, ensuring that peers have a higher expertise than the authors, the effectiveness of the inspection is likely to be higher. On the other hand, selecting peers with less or similar expertise when compared to authors can lead to a poorer effectiveness of the inspection process. Naturally, these quality considerations should be balanced with others, such as cost and schedule, as the usage of the best experts is likely to have a negative impact on these items, at least directly. However, this shortcoming can be compensated by decreased costs and schedule problems, as a result of the increased quality of the inspected artifacts.

As noted during this chapter, the effects of different expertise can be mitigating by using more sophisticated reading techniques during defect detection. So, the results of this experiment would not yield with reading by stepwise abstraction, or perspective-based reading.

6.7.3 Future work

As future work, we expect to expand on this experiment by exploring this interpretation of why two of our hypotheses were not confirmed. The deliverables of the project

that served as a basis for this experiment include some details that were not explored in this experiment, such as code complexity metrics, and the practitioners' assessment of the potential impact of the problems reported. We plan to further explore these data to strengthen the conclusions reported here and to explore other related hypotheses on the effect of expertise throughout the development process.

Chapter 7

Component reusability assessment

Contents

Background: *Eclipse Plug-ins development has achieved a considerable success, as user communities contribute with their own plug-ins to enhance the original platform's functionality. However, the real benefits for plug-in developers that result from providing extension points to their own plug-ins are unclear.*
Objective: *Our goal is to find out reuse patterns of Eclipse plug-ins to help deciding when is it worth spending resources providing extension points.*
Method: *We adopt a quantified single control group post-test only design, to contrast plug-ins from the basic Eclipse distribution with a set of other plug-ins, with respect to the availability and actual reuse of extension points.*
Results: *The results show that the vast majority of the extension points are not reused at all, except when those extension points are provided by the Eclipse platform.*
Limitations: *We validated the results for published Eclipse plug-ins. We should regard with caution inferences to a population including non-published plug-ins, or plug-ins for other platforms, and further validate such inferences.*
Conclusion: *The investment in providing extension points does not seem to have a direct return. Users are uncomfortable with building their software upon other plug-ins, unless those plug-ins are part of a major distribution (e.g. the Eclipse distribution).*

7.1 Motivation

7.1.1 Problem statement

As discussed in chapter 2, one of the main claims used to support the adoption of CBD is the alleged high return on investment (ROI) that one can achieve by building software from reusable components. The rationale is that it is less expensive to buy and adapt a reusable component than to develop its functionality from scratch. Commercially sponsored reports such as [Brooke 02] include claims on cost reductions attributable to the reuse of existing components to values of about 1/50 of the cost of building the reused components from scratch. Furthermore, the CBD community expects extensive reuse to lead to an increased quality of the component, because the component users' community feedback can be used as a driver for component's quality improvement efforts.

The account for the extent to which these benefits do occur, in practice, varies significantly throughout the literature. Nevertheless, several publications, including [Bass 01,Heineman 01,Szyperski 02,Crnkovic 02], refer to such benefits. A common point, also found in these references, is that component reuse has a potential for much greater benefits than those achieved so far.

In 1992, Nierstrasz *et al.* identified four main difficulties hampering the adoption of CBD [Nierstrasz 92]:

i. present day object-oriented languages do not fully support a component-oriented approach to software development;

ii. application development tools tend to emphasize programming and debugging rather than composition and reuse;

iii. it is difficult to abstract from acquired domain knowledge in order to engineer plug-compatible components for composing new applications;

iv. it is unclear how one can obtain a satisfactory return on the investment in reusable software components (from the point of view of a component user).

These difficulties were certainly a problem in the early 1990's. In spite of all the progresses on software development since then, they remain challenges for the community. Consider the problems **i.** and **ii.**, which have a predominantly technical nature: present day languages do not fully support CBD, as discussed in chapter 2 (e.g., with respect to behavior modeling, at a higher level of abstraction than that of source code). Reuse is becoming increasingly important but there is still a lot of emphasis on programming and debugging (even if we can perform these activities on top of more sophisticated programming languages, with extensive libraries of pre-fabricated classes and components).

In this chapter, we will explore problems **iii.** and **iv.**: the extent to which it remains difficult to abstract acquired domain knowledge to build plug-compatible components, and how difficult it remains to obtain a return on the investment in reusable software components.

Concerning this last point, Nierstrasz *et al.* claimed that *'existing software is reused only if it is part of the basic environment, if it is free, or if it constitutes a complete subsystem (such as a database). New approaches to software licensing and exchange of software information are needed if developers of reusable software are to receive a return on their investments.'* [Nierstrasz 92]

Do problems **iii.** and **iv.** remain unsolved, 16 years later?

The diversity of variants in current technologies used in CBD, each with its own strengths and shortcomings, may provide a blurred picture of the current patterns of reuse in CBD. As the different technologies could have a confounding effect in our study, we will narrow the study's focus to a particular approach to CBD: **plug-in based development**.

Plug-in based development has achieved a considerable success in the last decade. It is possible to to plug new components (plug-ins) into existing ones, adding to their functionalities. This form of incremental development through the integration of new plug-ins has achieved a widespread usage in several application categories, such as web browsers, or software development environments, ranging from open source freeware to commercial systems. The observational study presented in this chapter uses the concrete example of the Eclipse[1] plug-in architecture.

We examine a set of Eclipse plug-ins with respect to the effective usage of their extensibility mechanisms, to analyze the extent to which their extension points are used by other successful plug-ins. Which are the common characteristics of the plug-ins which get to be reused more often? Who reuses those plug-ins? Are there good incentives to provide extension points to plug-ins? How do Nierstrasz's remarks relate to the actual reuse profile of Eclipse plug-ins?

In order to address these concerns, it is useful to list a set of research questions (RQ), along with a short description of their rationale:

- **RQ1** It may be the case that, even if supported by component models, the reusability of components is not one of the main concerns when developing those components. If so, the likelihood of a plug-in providing extension points is fairly low, particularly if the plug-in is not part of the base plug-in framework (in this case, the standard Eclipse distribution). Both Szyperki's definition of software component as *"a unit of composition with contractually specified interfaces and explicit context dependencies only, **which is subject to third party composition**"* [Szyperski 02], and the plug-in based architecture, described in this chapter, emphasize the support that components and their component models provide for reusing

[1] `http://www.eclipse.org/`

components. However, Nierstrasz's remarks convey a concern on the challenge of making components reusable.

- **RQ2** If we consider only the plug-ins that provide extension points for our analysis, it may still be the case that the origin of the plug-in has a significant impact on the actual reuse of the plug-in. In other words, organizations may be willing to reuse their own plug-ins and plug-ins in the standard distribution of Eclipse, but not plug-ins provided by other sources. We break this hypothesis down into three sub-hypotheses:

  - **RQ2a** The extension points reuse ratio depends on the plug-ins origin. In this sub-hypothesis, we contrast plug-ins that are part of the standard Eclipse distribution with those which are not part of that distribution.
  - **RQ2b** In contrast with **RQ2a**, we can consider only the extensions to plug-ins extension points when the **extender plug-ins** are not part of the same product as the **extended plug-ins**. Again, we can test the reuse ratio of plug-ins from the basic distribution, *vs.* other plug-ins.
  - **RQ2c** Even when part of different products, plug-ins may have a common provider, and this may have an impact on their reuse profile. Therefore, it may be the case that plug-ins from the basic distribution are likely to have more external client products than other plug-ins.

The remainder of this chapter is dedicated to the pursuit of answers for these research questions, which will be used to define the set of research hypotheses, later in this chapter. We analyze Eclipse plug-ins available from a plug-in broker, in addition to the plug-ins distributed with Eclipse 3.2. With respect to the presentation of our study, we will follow the same structure of the previous chapter.

7.1.2 Research objectives

Our goal is to
analyze Eclipse plug-ins
for the purpose of characterizing the usage of the Eclipse plug-ins extension mechanisms
with respect to plug-ins reusability,
from the point of view of developers who may use the Eclipse plug-ins extension mechanism (in this case, the research team),
in the context of an observational study on Eclipse plug-ins available both from a broker and in the standard Eclipse distribution.

7.1.3 Context

This study uses publicly available Eclipse plug-ins, both from the Eclipse community and from the basic Eclipse distribution. We should consider the results of this study valid only in the context of Eclipse plug-ins, rather than as generic for other component models and technologies. We must conduct further research, with other component models, to check which conclusions are specific to the Eclipse plug-ins component model and which are generalizable to other models. The study includes a large sample of plug-ins with varied domains of application, complexity, state of development, license type, and developers. In that sense, the study's conclusions are generalizable to Eclipse plug-ins in general.

7.2 Related work

7.2.1 The Eclipse plug-ins architecture

Before presenting the remainder of this study, we discuss a few useful concepts concerning the domain from which we collected our sample: Eclipse plug-ins.

The Eclipse workbench is, essentially, a platform for software development tools. The Eclipse architecture is built upon the notion of plug-in. An **Eclipse plug-in** is a component that provides a service within the context of the *Eclipse workbench*. Except from a kernel component named *Eclipse Runtime*, the Eclipse workbench is built from a core set of plug-ins that, together, provide Eclipse's basic functionalities. The platform is open, in the sense that new plug-ins provided both by the platform developers and the Eclipse user community can be plugged into it. This combined effort for developing plug-ins makes Eclipse a versatile tool, due to the variety of extensions provided by thousands of available plug-ins.

To allow for this extensibility, each Eclipse plug-in may expose a set of configurable **extension points**. As a whole, the Eclipse platform provides a basic set of extension points that allows extending its functionality. Both the plug-ins available on the basic Eclipse release and those developed by the community (including tool vendors) may provide such extension points.

A plug-in extends the functionalities of another plug-in by adding an **extension** to one of the extension points provided by the other plug-in. New plug-ins are integrated on the workbench as extensions to existing plug-ins. The available extension points include the extension points provided by the basic Eclipse release, those made available by other plug-ins that have been added on to that basic release, and even the extension points provided by the plug-in that is being integrated in the platform.

Each plug-in is identified not only by its id, but also by its version. Plug-ins developed for Eclipse versions older than 3.2, also include information such as a human-readable plug-in name, the plug-in provider, as well as a reference to the set of runtime

libraries required by the plug-in, and the identity of the plug-in class that is used to instantiate the plug-in (a subclass of `org.eclipse.core.runtime.Plugin`). The Eclipse 3.2 plug-in manifest narrows the directly available information down to the specification of extension points, including identification info and an XML schema for additional data, and the specification of extensions. The latter include information concerning the identification of the extension point, as well as well-formed XML that conforms to the desired extension-point schema, which is used for configurating the extension.

In order to support our experimental work, we designed a metamodel for representing Eclipse plug-ins. Our metamodel is based on the Eclipse plug-ins manifest DTD. Figure 7.1 contains an excerpt of the Eclipse plug-ins metamodel, where we present the metaclasses which are relevant for this observational study. Several attributes and operations defined in the complete metamodel are hidden, to avoid cluttering the diagram.

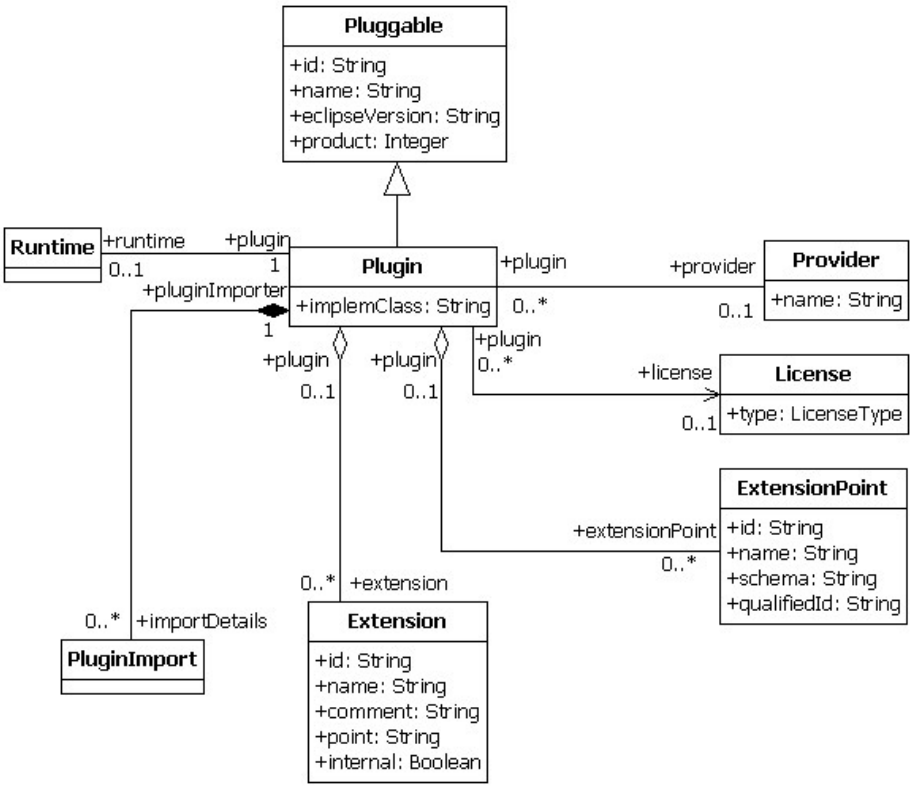


Figure 7.1: An excerpt of the Eclipse plug-ins metamodel

7.2.2 Experimental assessment of component reuse

In sections 2.5.5 and 2.5.6 of this dissertation, we discussed several metrics for CBD. The majority of those metrics was proposed to help assessing reuse in the context of CBD. This shows a great concern of the community in being able to assess reuse patterns in CBD. In a brief recapitulation of some of these proposals, we can observe how they relate to reusability:

- Bertoa *et al.* defined and validated metrics for COTS components, focusing on the usability of those components, as perceived by the component assemblers [Bertoa 04, Bertoa 06]. Their informal definitions relied in collecting information from the COTS's available documentation. The diversity of that documentation made the automation of their metrics collection process not practical.

- Dumke proposed a set of metrics for assessing the reusability of JavaBeans, which relied on white-box access to the components source code [Dumke 00]. This dependency on white-box access to components has the shortcoming of preventing the usage of this metrics set with black-box components.

- Washizaki *et al.* also proposed a metrics set to assess the reusability of components, and tested those metrics with JavaBeans [Washizaki 03]. Their proposal contrasted with Dumke's in several ways, namely in the fact that it supported the assessment of black-box components.

- Boxall and Araban focused on the textual complexity of component interfaces, as a measure of their understandability, which in turn is expected to influence the reusability of those components [Boxall 04].

All these proposals focused on metrics that were independent from the context of usage of each component. Other researchers presented proposals on metrics that rely on the context in which the plug-in is used for their computation:

- Seker proposed coupling and cohesion metrics for black-box components and component assemblies [Seker 04]. Coupling and cohesion metrics are also often related to the reusability of components, in the literature. For instance, a component with a high coupling to other components may be less desirable to reuse, because of the extra components that will have to be indirectly reused when choosing it.

- Hoek *et al.* proposed service utilization metrics that measured the effective reuse rate of component services [Hoek 03]. An interesting feature of these component-model independent metrics is that they contrast the potential utilization of a service with its actual reuse.

- Inoue *et al.* proposed a component significance ranking [Inoue 05], inspired by ranking algorithms by Google's page rank algorithm. The ranking represents the importance of a component in a potentially reusable population. Higher ranked components are components that are reused more often.

None of these proposals was made for Eclipse plug-ins, although some of them (especially Hoek *et al.*'s) can be adapted to the Eclipse plug-in architecture. Most of these metrics had a common concept underpinning their definition: there are some properties of a component, whether these correspond to better documentation, or easier to

understand interface, which are crucial for understanding the component's reusability. This is particularly noticeable in the first group of metrics presented here.

The second group of proposals adopts a more pragmatic approach to reuse. It assesses the actual reuse of components, using different strategies to achieve this goal, rather than trying to contribute to establishing some form of heuristics based assessment of component's reusability, when judging components in isolation.

The work presented in this chapter is closer to the one discussed in the second group. However, we will explore the reuse patterns of a special type of components - Eclipse plug-ins - using a different perspective than the one of our peers. Rather than trying to find a heuristic for making reusable components (as in the first group of metrics, or even in Seker's proposal) we will follow an approach which is inspired by the works of Hoek and Inoue. We borrow from Hoek *et al.* the basic idea of service utilization - the extent to which a service provided by a component is reused by any of the other components in the assembly (or conversely, the extent to which a service required by a component is available in the component assembly). The idea of a component ranking based on actual component reuse is borrowed from Inoue.

We will add to these ingredients a different perspective: the trust that component assemblers may or may not have in the component's origin.

7.3 Experimental design

7.3.1 Goals

The research objectives sketched in sub-section 7.1.2 are refined here as goals. Note that the perspective taken on this study and the context remain invariant for all the sub-goals, and are therefore represented by "(...)", as in the previous chapter, to highlight the differences between each sub-goal. Note that goal **G1** corresponds to research question **RQ1**, and so on.

G1:

Analyze plug-ins, **for the purpose of** their characterization, **with respect to** the extent to which belonging to the standard Eclipse distribution has an impact on the number of extension points they provide for reuse by other plug-ins, (...)

G2:

Analyze plug-ins,
for the purpose of their characterization,
with respect to the relationship between the origin of a plug-in (part of the standard Eclipse distribution *vs.* not part of it) and its reuse,
(...)

G2a:

Analyze plug-ins,
for the purpose of their characterization,
with respect to the relationship between the origin of a plug-in (part of the standard Eclipse distribution *vs.* not part of it) and its extension points reuse ratio,
(...)

G2b:

Analyze plug-ins,
for the purpose of their characterization,
with respect to the relationship between the origin of a plug-in (part of the standard Eclipse distribution *vs.* not part of it) and the ratio of extensions by plug-ins which are part of a different product than the one of the extended plug-in,
(...)

G2c:

Analyze plug-ins,
for the purpose of their characterization,
with respect to the relationship between the origin a plug-in (part of the standard Eclipse distribution *vs.* not part of it) and the number of different products from other providers in which there is at least one plug-in extending it,
(...)

7.3.2 Experimental units

The **theoretical population** of this observational study is the set of Eclipse plug-ins that are available to the Eclipse community, including the plug-ins made available by the basic Eclipse distribution. It would be unfeasible to identify and access all the eligible Eclipse plug-ins as such identification would imply being able to enumerate all the

plug-ins available in a particular moment in time. Instead, we use a representation of the population, known as **sampling frame**. Unlike the plug-ins in the theoretical population, the plug-ins belonging to the sampling frame can be enumerated.

Our sampling frame is the union of a set of Eclipse plug-ins made available by the Eclipse Foundation's plug-ins broker - Eclipse Plugin Central (EPC) [2] with the set of plug-ins available on the basic Eclipse distribution (Eclipse SDK 3.2.1). The plug-ins list made available through EPC is frequently updated, so we include in our frame all the Eclipse plug-ins which were listed by EPC on a particular moment in time (our frame was defined on the 30th of January, 2007, 15h00GMT). The sampling frame includes 788 plug-ins[3], ranging from open-source plug-ins to commercial ones. The plug-ins are distributed into 31 different categories, related to their respective domain of usage. These categories include 30 EPC categories and an extra category for the plug-ins in the standard distribution.

We consider this frame to be representative of the Eclipse plug-ins theoretical population, for the purposes of this study, as it is broad, in what concerns the domain of usage of the plug-ins. Furthermore, EPC is hosted by the Eclipse Foundation. The latter is the organization in charge of coordinating the development of the Eclipse workbench. Therefore, the EPC repository is a good representative of the plug-ins made available by (and to) the Eclipse community.

7.3.3 Experimental material

In this observational study, the experimental material corresponds to a sample taken from the sampling frame. The subjects are, in this case, the plug-ins under scrutiny. The sample taken from the sampling frame is a convenient sample. We only include plug-ins that are shipped as an archived file, but require no particular installation, other than copying the files to a specific directory in an Eclipse installation. In other words, our sample excludes the remaining plug-ins. Our analysis depends on knowing precisely which plug-ins are part of each plug-in bundle, so it is important for us to maintain a strict control on the origin of each plug-in. This strict control would conflict with installation wizards that are built so that the user can be oblivious of the plug-ins that they need to install.

7.3.4 Tasks

As noted on the previous section, the subjects of this study are Eclipse plug-ins. As such, this common item in the experimental design description is not applicable for this study.

[2] `http://www.eclipseplugincentral.com/`

[3] To be more precise, the sampling frame includes 788 plug-in bundles. While some of these bundles include a single plug-in, others include several plug-ins that support the plug-in we are installing.

7.3.5 Hypotheses and variables

Hypotheses

The observations on the problem statement section lead us to testing five different basic hypotheses, to assess the effect of plug-in origin, license type, application domain, as well as the plug-in's extension points availability and complexity, in the actual reuse rate of each plug-in. We identify the hypotheses as $H1$, $H2$, $H2a$, $H2b$, and $H2c$. For each of them, we now formulate both a null and an alternative hypothesis (e.g. $H1_0$ and $H1_1$). Our research hypotheses are as follows:

$H1_0$: A plug-in's origin has no significant impact on the number of extension points provided by that plug-in.
$H1_1$: A plug-in's origin has a significant impact on the number of extension points provided by that plug-in.

$H2_0$: A plug-in's origin has no significant impact on the plug-in's actual reuse.
$H2_1$: A plug-in's origin has a significant impact on the plug-in's actual reuse.

$H2a_0$: A plug-in's origin has no significant impact on that plug-in's extension points reuse ratio.
$H2a_1$: A plug-in's origin has a significant impact on that plug-in's extension points reuse ratio.

$H2b_0$: A plug-in's origin has no significant impact on that plug-in's external plug-ins clients ratio.
$H2b_1$: A plug-in's origin has a significant impact on that plug-in's external plug-ins clients ratio.

$H2c_0$: A plug-in's origin has no significant impact on that plug-in's number of external client products.
$H2c_1$: A plug-in's origin has a significant impact on that plug-in's number of external client products.

Independent variables

The independent variable is the same for all the hypotheses considered here. We will call it **Is part of Eclipse**. Its value is `true` for plug-ins which are part of the standard Eclipse distribution, or `false`, otherwise. In order to determine whether or not a plug-in is part of the standard Eclipse distribution, we compare its `product` instance variable with the integer constant `ECLIPSE_ID`, which corresponds to the unique product key

attributed to Eclipse when we built our product sample. In OCL, we can compute this value by defining the operation `IsPartOfEclipse` in the meta-class `Pluggable`, as in listing 7.1. Note that this meta-class, as well as others used in the formalization of metrics in this chapter, are part of the metamodel presented in figure 7.1.

Listing 7.1: Defining IsPartOfEclipse.

```
context Pluggable
  IsPartOfEclipse(): Boolean = self.product = ECLIPSE_ID
```

Dependent variables

The dependent variables in this study can be defined in the context of the meta-class `Plug-in`. We will start by providing an informal description of each of the variables. Then, we will formally define the OCL operations required to compute each of these dependent variables.

For hypothesis $H1$, the relevant variable is **Extension points**. This is a simple count of the extension points provided by a plug-in. Its value is an integer, and we can compute its value as in OCL using the `ExtensionPointsCount()` operation (listing 7.2).

Listing 7.2: Counting extension points.

```
context Plugin
  ExtensionPoints(): Set(ExtensionPoint) = self.extensionPoint
  ExtensionPointsCount(): Integer = self.ExtensionPoints()->size()
```

For hypothesis $H2a$, we will use, as auxiliary dependent variables, **Extension points**, as well as **UsedExtensionPoints**. The latter is a simple count of the extension points of a plug-in which are extended by any plug-in within our sample. These two variables are then used in the computation of our main dependent variable for this hypothesis: **Extension points reuse ratio**, which is a ratio of the used extension points, when compared to the available extension points of a plug-in. The two auxiliary variables are integers, while the ratio is a real number. In OCL, we will add operations `UsedExtensionPointsCount()` and `ExtensionPointsReuseRatio()` to the `Plugin` meta-class (listing 7.3).

Listing 7.3: Computing the reuse ratio of extension points.

```
context Plugin
  UsedExtensionPoints(): Set(ExtensionPoint) = ExtensionPoints()
    ->select(ep: ExtensionPoint | ep.IsUsed())
  UsedExtensionPointsCount(): Integer = UsedExtensionPoints()->size()
  ExtensionPointsReuseRatio(): Real =
    self.UsedExtensionPointsCount()/self.ExtensionPointsCount()
```

This definition requires an auxiliary operation `IsUsed()` to be defined in the meta-class `ExtensionPoint` (listing 7.4). This operation checks whether or not an extension

point's qualified identifier (a unique identifier for that extension point) is referred to as the extension point of any of the existing extensions in the sample.

Listing 7.4: Finding if an extension point is used.

```
context ExtensionPoint
  IsUsed(): Boolean = Extension.allInstances.point->includes(qualifiedId)
```

For hypothesis *H2b*, we will compute the count of **Client plug-ins**. This is the size of the set of plug-ins that extend at least one of the extension points provided by a plug-in. While some of these plug-ins may be part of the same product, others may belong to different products. We define **Client plug-ins from other products** as the size of the set of plug-ins that reuse a plug-in's extension point but are not part of the same product to which the extension point provider plug-in belongs. Both are integer values. The **External plug-in clients ratio**, computed by the operation `ExternalClientPluginsProvidersRatio()` is a real number corresponding to the percentage of **Client plug-ins from other products**, computed by `ExtensionPointsExternalClientPluginsCount()` in the total **Client plug-ins**, computed by `ExtensionPointsClientPluginsCount()`. These operations are presented in listing 7.5.

Listing 7.5: Computing the ratio of external plug-in clients.

```
context Plugin
  ExtensionPointsClientPlugins(): Set(Plugin) =
    self.ExtensionPoints().ClientPlugins()->asSet()
  ExtensionPointsClientPluginsCount(): Integer =
    self.ExtensionPointsClientPlugins()->size()

  ExtensionPointsExternalClientPluginsCount(): Integer =
    self.ExtensionPointsClientPlugins().product
      ->excluding(self.product)->size()

  ExternalClientPluginsProvidersRatio(): Real =
    ExtensionPointsExternalClientPluginsCount()
    ExtensionPointsClientPluginsProvidersCount()
```

These definitions rely on the definitions of the operation `ClientPlugins()`, made available in the meta-class `ExtensionPoint`. Listing 7.6 presents its definition.

Listing 7.6: Selecting an extension point's client plug-ins.

```
context ExtensionPoint
  ClientPlugins(): Set(Plugin) =
    Extension.allInstances->select(e : Extension |
      e.point = self.qualifiedId).plugin->asSet()->excluding(self.plugin)
```

Finally, for hypothesis *H2c*, we define the **number of external client products** as the size of the set of products that include plug-ins that extend any of the plug-in's

extension points. In order for a product to belong to this set, it must be provided by a different organization than the one which provided the extended plug-ins. This variable is an integer. The operation `ExternalClientProductsCount()` is defined in listing 7.7.

Listing 7.7: Counting the products that reuse a plug-in.

```
context Plugin
  ExternalClientProductsCount(): Integer =
    self.ExtensionPoints().ClientPluginProducts()->asSet()
      ->excluding(self.product)
```

This definition relies on the specification of `ClientPluginProducts()` in the meta-class `ExtensionPoint` (listing 7.8).

Listing 7.8: Selecting the client products of an extension point.

```
context ExtensionPoint
  ClientPluginProducts(): Set(Integer) =
    self.ClientPlugins().product->asSet()
```

7.3.6 Design

The design used in this observational study can be classified as a **quantified single control group post-test only design**, which can be presented as follows, using Trochim's notation [Trochim 06]:

```
N   O
N X O
```

This design results in 2 groups, one where the treatment (being part of the Eclipse standard distribution) is not applied, and the other where it is. In other words, the first group includes plug-ins which are not part of the Eclipse standard distribution, while the second one includes the plug-ins which are part of the sample.

The groups are non-equivalent. While we use all the plug-ins of the standard Eclipse distribution (version 3.2), we use a convenient sample of the plug-ins which are not part of that distribution, collected from an Eclipse plug-ins broker repository.

This design is applied in hypotheses $H1$, and $H2$ (including $H2a$, $H2b$, and $H2c$).

7.3.7 Procedure

The data collection required for this study consists in downloading the plug-ins and storing them in a consistent way, for further analysis. For each downloaded plug-in, we also store the meta-information made available by the plug-in broker. Some plug-ins are distributed as compressed archives (frequently, either as a *jar archive*, or as a *zip archive*). Others have fairly more sophisticated installers. Either way, installing a plug-

in in Eclipse includes copying a folder with the files defining the plug-in to a specific folder named *plugins* inside Eclipse's installation folder.

In our observational study, we analyze the contents of a replica of this folder, where we put not only the standard plug-ins but also the plug-ins downloaded from EPC. All compressed files are uncompressed, so that we can then extract information from them.

The tool support for this study includes both off-the-shelf components (the **USE tool**[4] and **SPSS**[5]) and custom made components (**Plugin2USE** and **PluginMetrics2SPSS**), developed for the purposes of this dissertation. Each component is implemented as a different, independent, tool. The architecture of our tool support follows the pipes and filters architectural style [Garlan 93].

The activity diagram in figure 7.2 details the activities and responsibilities of each of the components involved in the observational study. The first swim lane on the activity diagram represents the person(s) conducting the observational study. The second swim lane represents the repository used in this study. This corresponds to the file system. The third swim lane represents the Plugins2USE component, which we developed to support this study. The fourth swim lane represents one of reused off-the-shelf components: the USE tool. The fifth swim lane represents the second component we developed to support this experiment: PluginMetrics2SPSS. Finally, the last swim lane has the other reused off-the-shelf component: SPSS. If we consider the 4 components corresponding to the 4 swim lanes on the right, we can observe how the output of each component is used as input to the component immediately to its right.

The **experimenter** is responsible for setting up the study by collecting the data sample and creating the appropriate configuration for the components used in the data collection, transformation, and analysis. In this particular observational study, he collects Eclipse plug-ins, as provided by the plug-ins producers and stores them in a plug-ins repository (a simple directory on the file system). The experimenter is also responsible for creating three configuration items for the study. The metamodel representing the concepts relevant for this study, in UML, the specification of the metrics to be collected from that model's instantiation, using OCL, and the statistical treatment to be applied to the collected metrics, specified as an SPSS command file. Although such treatment can be carried out interactively by the experimenter using the statistics tool, storing the whole data treatment procedures in a script facilitates the replication of the observational study.

The plug-ins to be analyzed in the observational study are used as inputs to the **Plugin2USE** tool. We developed this tool to automatically extract information, as required by our study, from Eclipse plug-ins. The data used by the **Plugin2USE** component is the contents of the replica folder discussed earlier. This component will automatically collect the required information from this folder. Eclipse plug-ins include a **manifest file**, in XML, from which **Plugin2USE** extracts the information for our observational

[4] http://www.db.informatik.uni-bremen.de/projects/USE/
[5] http://www.spss.com/

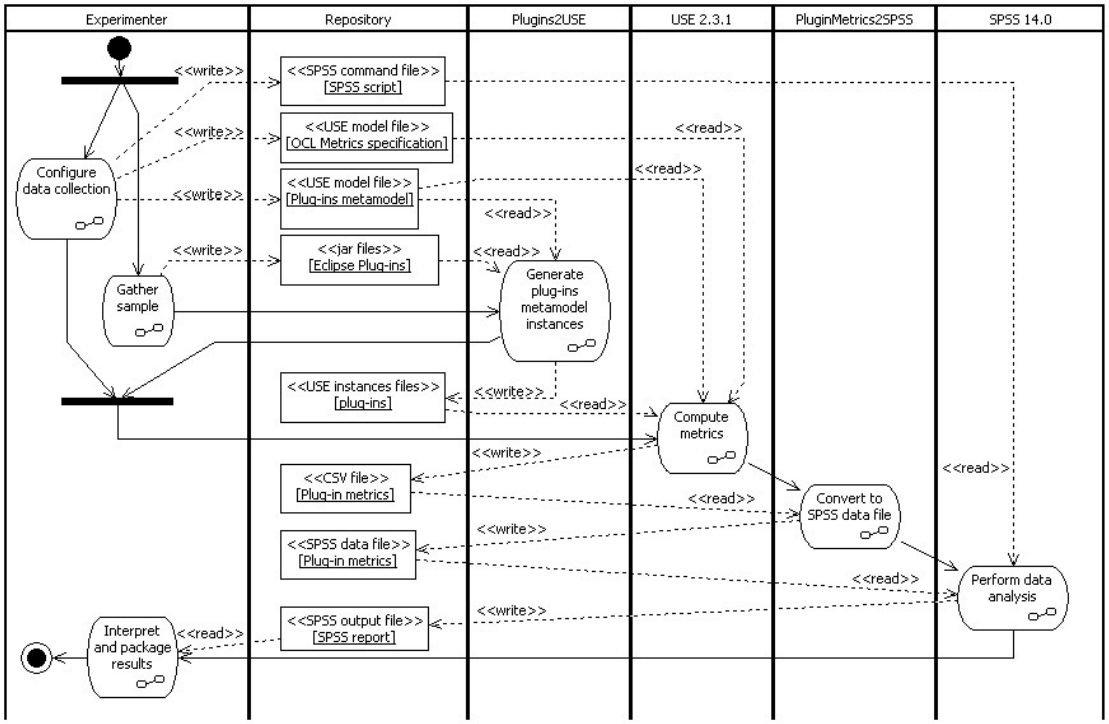


Figure 7.2: Data collection activities

study. **Plugin2USE** parses the plug-in manifest files an creates an instantiation of the plug-ins metamodel partially presented in figure 7.1. The instantiation is generated as a **USE** instances file. The instances file is a script, which is fed into the **USE** tool. The script contains **USE** commands to create all the meta-objects, the links among them, and to assign values to the attributes of those meta-objects.

The **USE** tool collects the metrics specified in OCL from the instantiated plug-ins. The metrics values are then stored in a text file, for further processing.

The **PluginMetrics2SPSS** component transforms the output of the **USE** tool into an SPSS data file.

Finally, we use **SPSS** to perform all the statistical tests required for testing our hypotheses. This statistics component allows scripting a sequence of commands for performing data analysis, thus contributing to the automation of the whole process.

7.3.8 Analysis procedure

We will follow these steps:

- **Descriptive statistics**: For all our independent and dependent variables, we will collect a set of descriptive statistics including the **mean**, **standard deviation**, the **minimum** value of the variable in the sample, the **maximum** value, the **skewness** and the **kurtosis** of the variable's distribution. These descriptive statistics will provide us with a first overview on our data, that we will further detail in subsequent analysis.

- **Data set reduction**: outliers and extreme values may be removed, if their presence biases our analysis.
- **Normality tests**: Data will be checked for normality, so that the statistics tests which are suitable for our data can be selected.
- **Analysis of differences between groups**: Finally, we will perform a test to detect whether there are significant differences between groups. This will allow us to test the hypotheses stated in section 7.3.5.

Each of these steps, and how they are followed in this study will be detailed in section 7.5.

7.4 Execution

7.4.1 Sample

We excluded from analysis any plug-ins which were not distributed as an archive, to avoid complicated installation procedures.

7.4.2 Preparation

The preparation of this observational study consisted, primarily, the development of the computational support required to perform the study. This includes the identification of two off-the-shelf components (USE and SPSS), as well as the development of the other two components (Plugin2Use and Use2SPSS), in our pipe and filter architecture. Both components were developed in C♯, for the .Net platform.

We conducted a pilot study on a small sample of plug-ins, to test our architecture with real data. Once we were satisfied with the performance of the developed tools and their interoperability with the off-the-shelf tools, we were ready for performing the data collection.

7.4.3 Data collection performed

In this observational study, data collection consisted in downloading a sample of Eclipse plug-ins made available in EPC. The main constraint in this task was the performance bottleneck imposed by one of our off-the-shelf components: with thousands of objects, and hundreds of thousands of links and attributes, the performance of the USE tool degrades significantly. Conducting this study with all the elements of the sampling frame was not feasible.

A secondary constraint concerned the extremely time consuming download of the plug-ins, as this task could not be conveniently automated - several plug-in producers

require filling registration forms before allowing the download. The option was to download as many plug-ins as possible, until the OCL tool's performance limitations made it unfeasible to increase the size of the sample, with the available resources.

The sample of plug-ins used in this study contains 588 plug-ins, which correspond to 32 plug-in bundles. Recall that the plug-ins, rather than the bundles, are the subjects in our sample.

7.5 Analysis

In order to facilitate the traceability between the data presented and the corresponding research hypotheses, each of the following sub-sections is labeled with the corresponding hypothesis identification, ranging from $H1$ to $H2c$. In all tables, when presenting the significance of the tests, we use the following typing convention: the significance of each test is highlighted in **bold** for tests with $p < 0,05$ and ***italic bold*** for tests with $p < 0,01$.

7.5.1 Descriptive statistics

For each variable, we present the number of cases, the mean value within our sample, the standard deviation, the minimum value, the maximum value, the skewness and the kurtosis.

H1

In order to assess $H1$, we will consider the **Extension points** variable. Table 7.1 summarizes its descriptive statistics.

Metric	N	Mean	Std. Dev.	Min.	Max.	Skewness	Kurtosis
Extension points	588	,87	2,537	0	41	8,321	111,175

Table 7.1: Descriptive statistics of the number of **Extension points**.

In order to decide whether or not it is appropriate to use parametric tests for our hypothesis, we need to check if the variable has a normal distribution. The positive skewness indicates an asymmetric distribution, with a higher frequency of the variable's lower values. In other words, the distribution is right-skewed. This contrasts with the normal distribution, which is symmetric and should therefore exhibit a skewness of 0.

Based on the value of the Kurtosis, we may say that the distribution is also **leptokurtic**. In other words, the probability of values being close to the mean is higher than in a normal distribution. The probability of having extreme values in the population is also higher than in a normal distribution.

Both the skewness and the kurtosis of the distribution provide us a hint on the non-normality of our data. We will use further tests to confirm the non-normality of this variable. Table 7.2 presents the results of two such tests: the **Kolmogorov-Smirnov with the Lilliefors correction** and the **Shapiro-Wilk**'s normality tests. The former is the most widely used test and adequate for our sample size. The latter is often used with smaller samples, and used here for confirmation purposes only. The null hypothesis, for each of the tests, is that the sample comes from a normal distribution. The alternative is that the sample comes from a non-normal distribution.

	Kolmogorov-Smirnov(a)			**Shapiro-Wilk**		
Metric	**Statistic**	**df**	**Sig.**	**Statistic**	**df**	**Sig.**
Extension points	,366	588	***,000***	,361	588	***,000***

Table 7.2: Normality tests for the **Extension points** variable.

In conclusion, we cannot assume the sample to come from a normal distribution. We will have to use non-parametric tests to assess hypothesis $H1$.

$H2$ (including $H2a$, $H2b$ and $H2c$)

Several of the plug-ins in the sample provide no extension points. While this information was, by itself, interesting for assessing hypothesis $H1$, it can be argued that having in our sample plug-ins which provide no extension points may be a confounding effect for further analyzing the reuse patterns of the plug-ins which were built for reuse. If we exclude from the sample all the plug-ins with no extension points, we are left with 163 plug-ins. In order to assess hypothesis $H2$, we will consider only the 163 plug-ins which provide extension points. Hypothesis $H2$ concerns whether the origin of the plug-in has an effect on its actual reuse.

It should be noted that filtering out the plug-ins which do not provide any extension points has no effect on the metrics of any of the remaining plug-ins, because this exclusion is performed **after** computing the metrics, rather than before. This is a perhaps subtle, but essential detail: if we were to exclude the plug-ins **before** computing the metrics, a plug-in that does not provide any extension points would no longer be accounted for as a client, when extending another plug-in (which could impact metrics that depend on the client plug-ins, such as the number of **used extension points**).

Table 7.3 summarizes the descriptive statistics for the dependent variables considered while testing this hypothesis. Table 7.4 presents the corresponding normality tests. All the variables have non normal distributions. We have to use non-parametric tests for hypotheses $H2a$, $H2b$ and $H2c$.

7.5.2 Data set reduction

Apart from the selection of a subset of the sample for the analysis of hypothesis $H2$, no data set reduction is performed. Although there are outlier and extreme values in

H	Metric	N	Mean	Std. Dev.	Min.	Max.	Skew.	Kurt.
H2a	**Extension points**	163	3,13	4,025	1	41	5,757	48,682
	Used extension points	163	2,44	3,616	0	37	6,077	52,288
	Extension points reuse ratio	163	,7773	,35178	,00	1,00	-1,399	,445
H2b	**Client plug-ins**	163	7,53	24,607	0	267	8,264	80,456
	Client plug-ins from other products	163	4,06	19,813	0	224	9,215	96,841
	External plug-in clients ratio	110	,1849	,29843	,00	1,00	1,379	,429
H2c	**Number of external client products**	163	,84	2,815	0	26	6,017	44,271

Table 7.3: Descriptive statistics for the filtered sample, where the plug-ins with no extension points are excluded.

		Kolmogorov-Smirnov(a)			Shapiro-Wilk		
H	**Metric**	**Statistic**	**df**	**Sig.**	**Statistic**	**df**	**Sig.**
H2a	**Extension points**	,298	163	***,000***	,513	163	***,000***
	Used extension points	,250	163	***,000***	,503	163	***,000***
	Extension points reuse ratio	,344	163	***,000***	,658	163	***,000***
H2b	**Client plug-ins**	,380	163	***,000***	,282	163	***,000***
	Client plug-ins from other products	,419	163	***,000***	,198	163	***,000***
	External plug-in clients ratio	,369	110	***,000***	,666	110	***,000***
H2c	**Number of external client products**	,383	163	***,000***	,325	163	***,000***

Table 7.4: Normality tests for the metrics, after filtering out the cases where no extension points are provided. (a) stands for Lilliefors significance correction.

the samples (both the complete and the one used in hypothesis $H2$), we choose not remove them. The outlier and extreme values in the sample are not attributable to any problems in the data collection process. Removing them would be an unnecessary threat to the validity of the obtained results, as it would eliminate information that we consider relevant for our analysis. For instance, if a particular plug-in is extended much more often than the others, it will be marked as an outlier, or extreme value. Such a plug-in can be considered a hub component, in the Eclipse architecture. Eliminating it from the sample would sacrifice this information, which seems relevant, particularly when our goal is precisely to characterize the features of the plug-ins which get to be extended.

7.5.3 Hypotheses testing

H1

Hypothesis $H1$ concerns whether or not there are significant differences between the usage of the extension points mechanism, between plug-ins from the basic Eclipse distribution, when compared to other plug-ins. Our independent variable is **IsPartOfEclipse**, and we use it to discriminate between the two samples (the plug-ins in the standard Eclipse distribution *vs.* all the other plug-ins in the sample).

In order to compare the usage of this mechanism, we will perform the **Mann-Whitney U** test (table 7.6), which is a non-parametric alternative to assess whether two samples of observations come from the same population. The test starts by rank-

ing all the observations, regardless of the sample they come from. Values are ranked in descending order. Table 7.5 summarizes the information concerning the computed ranks.

	Is part of Eclipse	N	Mean Rank	Sum of Ranks
Extension points	false	462	279,74	129239,50
	true	126	348,62	43926,50

Table 7.5: Ranks for **H1**

As we can see, only 126 of the analyzed plug-ins come from the standard Eclipse distribution, and they have a lower mean rank (the lowest ranks correspond to the highest values). If the distributions come from the same sample, they have equal probability distributions. The Mann-Whitney test summarized in table 7.6 leads us to reject the null hypothesis (the two samples come from the same population). There is a high probability they come from different samples.

	Mann-Whitney U	Wilcoxon W	Z	Asymp. Sig. (2-tailed)
Extension points	22286,500	129239,500	-5,122	***,000***

Table 7.6: Mann-Whitney U test for the **Extension points variable**. The grouping variable is **IsPartOfEclipse**.

The test's results are confirmed with a **Two-Sample Kolmogorov-Smirnov** test (table 7.7). This test relies on the same rank classification (already presented in table 7.5). Its significance confirms the results presented for the Mann-Whitney U test.

	Most Extreme Differences			Kolmogorov-Smirnov Z	Asymp. Sig. (2-tailed)
	Absolute	Positive	Negative		
Extension points	,213	,213	,000	2,118	***,000***

Table 7.7: Two-Sample Kolmogorov-Smirnov test for **Extension points**

These results indicate that **the authors of plug-ins which are not part of the Eclipse standard distribution provide significantly less extension points than those available in the plug-ins of the standard distribution**.

At this point of our analysis, it is tempting to go back at our experimental design and add an extra hypothesis concerning the distribution of the existing extension points in the plug-ins which are not part of the standard distribution. The information collected so far indicates that most of the plug-ins which are not part of the standard Eclipse distributions provide no extension points. But adding a shallow hypothesis concerning that percentage is not particularly interesting. Instead, we can complement our test of hypothesis $H1$ with the following extra information, conveyed by figure 7.3: the distribution of the plug-ins which are not part of the standard Eclipse distribution, according to the number of provided extension points (in the horizontal scale), is similar to a Pareto distribution, where close to 77% of the plug-ins provide no extensions. The horizontal line drawn at 80% provides a visual hint to remind us that the Pareto

distribution is often associated with the *80-20 rule of thumb*, which states that 80% of the effects (available extension points) come from 20% of the causes (plug-ins providing those extension points). In this sample, all the extension points are provided by around 23% of the plug-ins. The monotonic function represented by the non-decreasing line on the top represents the accumulated sum of all the previous frequencies. The scale on the left side represents the percentage of plug-ins, while the one on the right side represents the corresponding count of plug-ins.

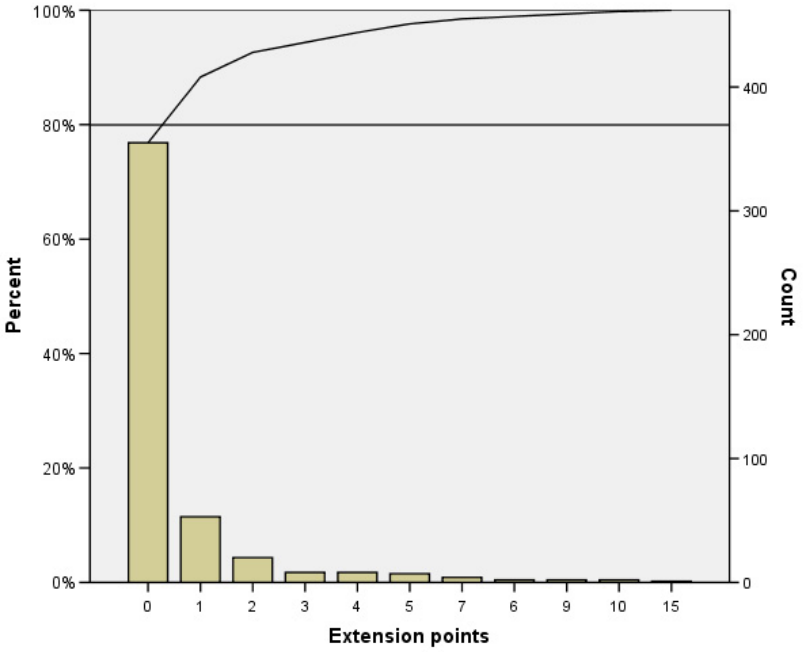


Figure 7.3: Extension points distribution, for plug-ins which are not part of the standard Eclipse distribution.

H2

Hypothesis $H2$ concerns whether or not the origin of a plug-in is a key factor with respect to the plug-in's reuse. We will look into this hypothesis using 3 perspectives: the ratio of plug-ins which provide extension points which are extended by other plug-ins, the ratio of plug-ins which are extended by plug-ins which are not produced by the same provider, and the number of products which reuse the plug-in (not counting with the product of which the plug-in is part of).

As with hypothesis $H1$, we will perform the Mann-Whitney test, and confirm the results with the Kolmogorov-Smirnov test. Table 7.8 presents the ranks computed for each of the variables, which will be used in both tests. An extra column **H** was added on the left of these tables, to identify the sub-hypotheses under scrutiny. For each of these sub-hypotheses, the decisive test is the last. Both with $H2a$ and $H2b$ we also present the tests for a pair of auxiliary variables, as this will help in the interpretation of results.

H	Metric	Is part of Eclipse	N	Mean Rank	Sum of Ranks
H2a	**Extension points**	false	107	76,16	8149,50
		true	56	93,15	5216,50
	Used extension points	false	107	75,21	8048,00
		true	56	94,96	5318,00
	Extension points reuse ratio	false	107	82,91	8871,00
		true	56	80,27	4495,00
H2b	**Client plug-ins**	false	107	72,34	7740,50
		true	56	100,46	5625,50
	Client plug-ins from other products	false	107	69,63	7450,00
		true	56	105,64	5916,00
	External plug-in clients ratio	false	66	42,58	2810,50
		true	44	74,88	3294,50
H2c	**Number of external client products**	false	107	69,53	7440,00
		true	56	105,82	5926,00

Table 7.8: Ranks for **H2**

The **Mann-Whitney test** summarized in table 7.9 compares the distribution of each of the variables, contrasting two groups: the plug-ins which are part of the standard Eclipse distribution, and those which are not. All the variables but the **extension points reuse ratio** show significant differences among the two groups.

H	Metric	Mann-Whitney U	Wilcoxon W	Z	A.Sig.(2-t)
H2a	**Extension points**	2371,500	8149,500	-2,296	***,022***
	Used extension points	2270,000	8048,000	-2,637	***,008***
	Extension points reuse ratio	2899,000	4495,000	-,385	,700
H2b	**Client plug-ins**	1962,500	7740,500	-3,693	***,000***
	Client plug-ins from other products	1672,000	7450,000	-6,128	***,000***
	External plug-in clients ratio	599,500	2810,500	-6,037	***,000***
H2c	**Number of external client products**	1662,000	7440,000	-6,185	***,000***

Table 7.9: Mann-Whitney U test. The grouping variable is **IsPartOfEclipse**. **A.Sig.(2-t)** stands for **Two tailed asymptotic significance**.

Table 7.10 presents the results of the Kolmogorov-Smirnov test. In this case, we cannot conclude that three of the variables (**extension points**, **extension points reuse ratio**, and **client plug-ins from other products**) came from samples with a different distribution.

		Most Extreme Differences				
H	**Metric**	**Absolute**	**Positive**	**Negative**	**K-S Z**	**A.Sig.(2-t)**
H2a	Extension points	,200	,200	,000	1,213	,105
	Used extension points	,277	,277	,000	1,677	***,007***
	Extension points reuse ratio	,098	,098	-,091	,597	,868
H2b	**Client plug-ins**	,293	,293	,000	1,775	***,004***
	Client plug-ins from other products	,415	,415	,000	2,516	***,000***
	External plug-in clients ratio	,591	,591	,000	3,036	***,000***
H2c	**Number of external client products**	,415	,415	,000	2,516	***,000***

Table 7.10: Kolmogorov-Smirnov test. **K-S Z** stands for **Kolmogorov-Smirnov Z**. **A.Sig.(2-t)** stands for **Two tailed asymptotic significance**. The grouping variable is **IsPartOfEclipse**.

We can aggregate the results of the Mann-Whitney U and the Kolmogorov-Smirnov tests and adopt a conservative position to analyze their results. If we can reject the null hypothesis using both tests, we will do so. Otherwise, we will not reject it.

Starting with the tests relevant for sub-hypothesis $H2a$, the only difference which is considered significant by both tests concerns the total number of extension points of a plug-in which get to be reused. **The higher mean rank of the plug-ins which are part of Eclipse indicates that the extension points in these plug-ins are more reused.** The conflicting results concerning the number of extension points provided by the plug-ins in the two groups, along with the similar reuse ratio of extension points in the two groups, lead us to think that the observed difference in the number of reused extension points is likely to result mainly from a higher number of extension points being made available. In conclusion, we cannot reject $H2a_0$.

Concerning $H2b$, we reject the null hypothesis $H2b_0$ and accept its alternative. The data indicates that **there is a significantly higher number of client plug-ins, both when we include all client plug-ins and when we only include the client plug-ins that are part of other products**. When combining these two variables in the external plug-ins reuse ratio, we still obtain a significant difference between the two groups. **The plug-ins in the Eclipse distribution are relatively more extended by plug-ins from other products than other plug-ins**.

Finally, in $H2c$, we reject the null hypothesis $H2c_0$ and accept the alternative: **the number of external client products is clearly higher for plug-ins which are part of the standard Eclipse distribution**.

7.6 Interpretation

7.6.1 Evaluation of results and implications

H1

Although, from an architectural point of view, the plug-ins in the Eclipse standard distribution are potentially similar to those that can be added to them by the community, **being part of the standard distribution does seem to have implications in plug-ins design choices, in practice. Plug-ins from the standard distribution provide extension points more often than those which are not part of it**. In fact, a large majority of the plug-ins which are not part of the Eclipse standard distribution provide no extension points at all. In other words, they are designed to be plugged to the platform, but provide no plugs for other plug-ins to extend them.

H2

Once we filter out the plug-ins that provide no extension points, we no longer observe a significant difference in the percentage of extension points which are reused, when comparing plug-ins from the standard Eclipse distribution with other plug-ins ($H2a$). Although there is not a significant difference concerning the extension points reuse ratio, there is a significant difference in the mean number of extension points which are actually extended.

In order to understand why the results in $H2a$ occur, consider the following: the Eclipse platform is built from plug-ins which extend other plug-ins within the standard distribution. In addition, plug-ins, which are not part of the standard distribution, may also extend the standard plug-ins. This may explain why the count of extension points which are reused is higher, in general, when considering plug-ins from the standard distribution.

Concerning $H2b$, there is a significantly higher ratio of client plug-ins from other products that use the extension points of the plug-ins in the Eclipse distribution, when compared to using extension points from other plug-ins. Both the numerator (client plug-ins from other products) and the denominator of that ratio (all client plug-ins) are significantly higher for plug-ins in the Eclipse distribution. The difference between groups is greater when we are considering client plug-ins from other products, than when we are considering all client plug-ins, thus leading to the significant difference in the ratio of external plug-in clients, as well.

Finally, for hypothesis $H2c$, we observe a clearly different mean number of external client products, extending the extension points. The Eclipse plug-ins from the standard distribution are extended much more products from other providers than other plug-ins, if we just consider the extensions made by other products.

In summary, although there is not much difference in the reuse ratio of plug-ins's extension points, we observe that **the origin of the plug-ins being reused is closely related to the kind of plug-ins that reuse them**. The likelihood of a plug-in's extension points being used by plug-ins which are not part of the same product is much higher if the plug-in providing the extension points is part of the basic platform. Furthermore, this effect is also observable when we contrast the extensions made by a third party to plug-in extension points. Third party extension is more frequent when the extension points are located in the standard Eclipse distribution.

7.6.2 Threats to validity

Internal validity threats

In what concerns the **internal validity of the study**, we consider two sorts of validity threats: social threats and multiple group threats.

Social threats to internal validity could stem from the usage of differentiated treatments within our sample. We can safely dismiss this threat. The treatment in this observational study concerns being part of the Eclipse standard distribution, *vs.* not being part of it. The study itself has no influence on the behavior of the plug-in producers as it is designed and carried out after the publication of the plug-ins, thus being unable to influence the analyzed plug-ins and their producers options while building them in any way.

We can dismiss the potential effect of single group threats as well, because we are using a control group in this study. We should, however, consider potential **multiple group threats**:

- **History**. To the best of our knowledge, no external event had a significant influence in the results of our tests. In particular, we are not aware of events that might have influenced the two groups differently. An example would be if some evolution of the plug-in design process made a difference among older and more recent plug-ins. However, we have no reason to believe that such evolution would affect differently the two groups under scrutiny.

- **Maturation**. We are only using a single version of each plug-in, so we can exclude pre-test/post-test evolutions. A possible maturation effect that could occur relates to the stage in the development process of the plug-ins in both groups. It is possible that the plug-ins in the Eclipse distribution are, on average, more mature than the other ones, with respect to their development process. However, as all the plug-ins under scrutiny are publicly available and they are, in general, considered by the plug-ins broker as stable, or at least in beta versions, we expect any maturation effects to be mitigated.

- **Testing**. This threat does not apply to our chosen post-test only experimental design. There is no repetition of activities during the experiment, thus avoiding this validity threat.

- **Instrumentation**. The same instrumentation was used with all the plug-ins in both groups, so it seems unlikely that it may have introduced some sort of differentiation among the groups.

- **Statistical regression**. This is not applicable to our design, as we used no pre-test information in the selection of the plug-ins in both groups. This threat should be considered if we were to perform a pre-test and then use its results to establish a selection criterion to choose only the plug-ins with a high pre-test score, or only those with a low pre-test score.

- **Selection**. The fact that our design uses non-equivalent groups makes this a potential threat. While the group of plug-ins from the standard Eclipse distribution

includes all the plug-ins in the selected Eclipse version, it is virtually unfeasible to create a truly random sample, or, for that matter, to enumerate the whole population) of plug-ins which are not part of the sample. We are only using plug-ins which were made available through a plug-ins broker (the most important one, in the Eclipse community). This excludes all the plug-ins which were not available at the time of our sample collection, in that broker's list. Sampling was made through convenient sample, which has also the potential to introduce some sort of bias, even if the researcher is not aware of it. All these factors considered, we did our best efforts to avoid validity problems which might result from the selection of plug-ins in our sample, by using plug-ins with varied domains of application, size, license types, and so on.

- **Mortality**. This threat did not apply to our observational study, as none of the plug-ins available in our sample was excluded from the analysis.

- **Ambiguity about direction of causal influence**. This is a potential threat in the sense that if a plug-in which is not part of the standard Eclipse distribution becomes a hub component in many plug-in based solutions, it may be the case that, in subsequent versions of the Eclipse platform, the Eclipse community integrates that plug-in as part of the standard distribution.

External validity threats

External validity refers to our ability of generalizing results beyond the scope of this experiment. We consider three potential sources of threats:

- **People**. This threat is not directly applicable to our design, because our subjects are plug-ins, rather than people (e.g. the plug-in developers).

- **Setting**. All the plug-ins used in this study are publicly available. Some of the reuse patterns observed here may not necessarily hold when considering plug-ins developed for personal, non-commercial use. For instance, it is plausible that a plug-in which exposes extension points, but has a license that only allows free of charge extension for personal use, may be extended in that context, but not by other products in the market. Another setting concern relates to the focus of our attention on the usage of a particular Eclipse plug-in extension mechanism: extension points. These can be compared to UML ports with provided interfaces, for instance. However, the findings for Eclipse plug-ins are not necessarily valid for plug-ins in other plug-in based architectures, or using different extension mechanisms. For instance, it is often the case that, in order to reduce coupling between components, several components are able to interoperate with each other by using a common data interchange protocol. For instance, several UML editor components are able to interchange information among them using

XMI. This kind of reuse is not addressed in our study. Finally, extrapolations to other component models would require validation with components of those models.

- **Time**. It may be the case that future developments of plug-in architectures may render the observations reported here as obsolete. In fact, for supporters of a more developed component-market-oriented view of CBD, this seems not only possible, but desirable. So, these observations should be considered in the context of a generation of plug-in products that use this extension mechanism. Extrapolations to other contexts should be done with caution.

Construct validity threats

With respect to construct validity threats, we consider two categories: social and design threats. Typical social threats include:

- **Hypothesis guessing**. This is not applicable to our observational study. Plug-in developers were not aware of this study while developing and making their plug-ins available.
- **Evaluation apprehension**. For the same motive as in hypothesis guessing, we do not regard this as an applicable threat to this study. Plug-in developers would not change their development options (e.g. by including more extension points in their plug-ins) due to concerns on the outcome of an assessment they were not aware of, while developing their plug-ins.
- **Experimenter's expectancies**. Our expectancies while performing this study had no influence in the observed results, as, again, we had no implicit or explicit influence in the options of plug-in developers concerning the usage of plug-in extension points.

Construct validity design threats include:

- **Inadequate pre-operational explication of constructs**. Our approach to metrics definition supports a rigorous definition of all the metrics in this experiment. We regard this as an effective mitigation of this potential threat.
- **Mono-operation bias**. This threat is mitigated by using a fairly large sample of plug-ins in each of the groups. This contrasts with comparing, for example, one plug-in from the Eclipse distribution with an external plug-in and inferencing from there. This limits the inference scope to Eclipse plug-ins. The threat does exist (and is not mitigated by our design), if one extrapolates from the results obtained here to other extension mechanisms and component models. Such extrapolations would have to be validated (e.g. through differentiated replicas of our study).

- **Mono-method bias**. This threat results from using a single kind of measure, or observation. Our study tries to mitigate this threat by using more than one way of measuring the usage of extensions among plug-ins. As all our measurements are based in the usage of plug-in extension points, other forms of dependencies among plug-ins are not considered here.

- **Confounding constructs and level of constructs**. It follows from the previous threat, that plug-in reusability is being measured indirectly using a specific mechanism (extension points). Differentiated replicas of this study could use other mechanisms (e.g. cooperation through common data repository sharing).

- **Interaction of different treatments**. As a single treatment is involved in our experiment, this threat is does not occur in our design.

- **Interaction of testing and treatment**. This does not apply to our design, particularly as this is an observational study.

- **Restricted generalizability across constructs**. This relates to the presence of undesired side-effects that would results as an indirect outcome of providing or extending extension points. The data available in this study does not help us to speculate much on this matter. An example would be if we were to find evidence that having an extension point extended had a negative impact for the plug-in's producers.

Conclusion validity threats

The potential threats to conclusion validity include:

- **Statistical power** The usage of non-parametric tests, advisable from the results in our early data analysis, leads to a lower statistical power than the one which would be achievable using parametric tests. The non-normality of our data determined this constraint.

- **Violated assumptions of statistical tests**. To the best of our knowledge, all the assumptions of the used statistical methods were met.

- **Fishing and the error rate**. When too many tests are performed, some of them may reveal spurious relations between variables, purely by chance. With the number of tests performed in this study, we have no reason to think the relations found between variables are spurious.

- **Reliability of measures**. One of the metrics used in our study is the number of used extension points, which is computed by counting the number of extension points used by any of the plug-ins in our sample. If we were to add extra plug-ins to the sample, it is possible that previously unused extension points would

become used by the new plug-ins. With almost 600 plug-ins in our sample, we are confident this would have no major effect in the general conclusion of this study, although this limitation does constitute a threat to the reliability of this measure.

- **Reliability of treatment implementation**. The observational study was conducted by a single person (the author of this dissertation), with the usage of automated tools. This uniformity mitigates this potential threat.

- **Random irrelevances in experimental setting**. This threat refers to features in the experimental setting which might create sources of variation in the outcome of tests, even if those features are irrelevant to the tests being performed. To the best of our knowledge, this threat was not present in our setting.

- **Random heterogeneity of subjects**. The fairly large number of plug-ins in the sample mitigated this threat.

7.6.3 Inferences

The analysis performed in this observational study should hold for larger samples of Eclipse plug-ins. In other words, we expect the same patterns of extension to be observable with other plug-ins which are publicly made available through plug-in brokers.

It is less clear whether or not all these observations apply to Eclipse plug-ins which do not get to be available in brokers. We expect some of the properties to hold, namely those relating to not investing in providing extension points to other plug-in producers. But we can imagine scenarios where this would not hold. For instance, during visits to poster sessions in software engineering conferences, we have noted that many graduate students develop Eclipse plug-ins as part of their research. It may be the case that these developers extend plug-ins made available through the broker, but which are not part of the standard distribution. They may also provide extension points for other colleagues to build on their work. These potential differences, that we are not able to assess from our sample, would result, to a certain extent, from the frequent differentiation concerning the inclusion of components in applications for personal use, *vs.* in commercial applications. The former reuse is often free, unlike the latter.

Extrapolating the results of this study to other component models is also risky. For some domains, such as graphical user interfaces development, the component market seems to be more developed, so we would expect to find a higher proportion of reuse of components by independent producers in those domains.

Overall, we think that the basic pattern of component reuse through extension is still mostly centered around a fairly limited set of trusted component frameworks, rather than on a true component market where alternative components for the same task are made available and reused. Note that this market does exist for components

which are *"final"* products themselves. Our comment relates to those components which would be reused as part of larger applications.

7.6.4 Lessons learned

In this section, we will discuss the lessons learned with respect to the operationalization of the observational study itself, rather than discussing the implications of the study's results. There were several challenges to overcome, to ensure the data quality for our analysis.

Although perhaps regarded as a "minor", or "less noble" activity, data collection often represents a large percentage of the effort while conducting a study such as the one reported here. Consider the effort of downloading and preparing the data for analysis, so that all plug-ins were stored in a uniform way, before being fed into our data collection pipeline. When feasible, using scripts for automating the repetitive tasks is highly advisable. However, there is some variety of formats in which the plug-ins are made available, which implies a somewhat repetitive effort to verify they are stored uniformly. Although it is tempting to build a web crawler to perform the downloads, this was not feasible in this study, as most plug-ins required the downloaders to fill in registration forms, before allowing the downloads. In a nutshell, the effort required for data collection must not be neglected.

The sheer volume of information gathered for analysis, with over 250 Megabytes of plug-in's distributions to explore was a problem. If we revisit figure 7.2, we can see how our data is processed by a pipeline of applications to support our analysis. The **Plugins2Use** component generated files containing the **USE** instances which were then fed to the **USE** tool. The **USE** tool turned out to be the bottleneck in our architecture. While none of the remaining tools showed any relevant efficiency limitations for their tasks, the **USE** tool stores its data structures in main memory, rather than having a database in the background where it can get and store information. With over 50 Mb of instantiation files [6], the memory-based data structures of the **USE** tool turned out to be very inefficient in the data loading process. Loading the whole information base took around 45 minutes in a Intel Core2 Duo T7200 (2x 2.20GHz), with 2GB of DDR2 RAM. The bottleneck lies precisely in the data loading process. Collecting the metrics used in this study, once the tool has all the necessary data loaded, is relatively fast. It took less than 1 minute to compute the results of all the necessary queries.

Efficiency is not a major concern for this kind of observational study, as we were not aiming to have this information in real time. If we were to integrate this sort of information in an IDE, it seems unlikely that we would require this volume of information to be processed in real time. Nevertheless, with around 600 plug-ins and the corresponding tens of thousands of objects and links, the OCL evaluation tool used

[6] The **USE** instantiation files are text files with instructions to create objects, assign values to those object's instance variables, and create links among those objects.

in this dissertation showed scalability problems. So, when conducting studies of this, or larger magnitudes, it would be advisable to use an OCL evaluation tool supported with a persistent data repository, which would allow a more efficient access to data.

7.7 Conclusions and future work

7.7.1 Summary

In this chapter, we report on an observational study where we analyzed the extension patterns in Eclipse plug-ins. We used a quantified single control group post-test only design. This allowed us to contrast the usage of the plug-in extension mechanism of **extension points** with two groups of plug-ins: the plug-ins from the standard Eclipse distribution, and a sample of plublicly available plug-ins, collected from a plug-in broker.

The results obtained in this study indicate that Eclipse plug-ins have different extensibility patterns when we contrast both groups. Plug-ins from the standard distribution are much more likely to be extended by external developers than other plug-ins.

The extension relationship between plug-ins can be represented as a directed graph, where the nodes represent plug-ins and the edges are directed from the extender plug-in to the extended plug-in. The nodes representing plug-ins which are not part of the standard distribution will have outgoing edges, but, frequently, no incoming edges. This effect is not very clear when we consider all the edges in the graph, because it includes edges that link plug-ins provided as part of the same product. It is common for products to be implemented by several plug-ins, and normal to find that those plug-ins are linked. So, the plug-in's extensibility mechanism is used in the composition of plug-ins developed in-house, or belonging to the basic platform. The contrast against the plug-ins from the basic platform becomes apparent when we exclude from the graph all the edges that are linking plug-ins from the same product, and even more evident when we exclude the edges linking nodes which represent plug-ins developed by the same provider.

Our sample of the universe of Eclipse plug-ins excludes those which are not made publicly available through the main Eclipse plug-in's broker. Although we believe this constraint allows us to have a representative sample of mature plug-ins, it excludes plug-ins which are not developed with the purpose of being made available to a wide audience, such as plug-ins built for personal usage. Another limitation of this study is that it is unfeasible to collect all the plug-ins available and use them as input for our study. As, along with other dimensions, we are measuring the observed usage of the plug-ins extension mechanism, it is possible that the actual reuse rate is higher than our estimate.

Nevertheless, the size of our sample and the obtained results lead us to think the

main conclusions would apply to the population of publicly available Eclipse plug-ins.

7.7.2 Impact

One of the goals of the CBD community is to strive for trustworthy components upon which developers would be willing to produce their own more sophisticated components (in this case, plug-ins). The observations in our study indicate that a *"quality seal"* of belonging to the basic platform remains a very important asset, when building components.

We observed that most of the analyzed components which were not part of the standard distribution were not built to be further extended by a third party. This information hints us on the lack of incentives of software producers to spend their resources to provide extension points that are not likely to be widely reused. This hint is reinforced by the lack of external reuse observed in the majority of plug-ins which do provide extension points. The exceptions to this lack of external reuse come precisely from the plug-ins in the basic platform. It is possible that this leads to a self feeding pattern, where the lack of demand for extension points in plug-ins which are not part of the standard is an incentive for plug-in producers not to develop such extension points. In turn, this does not foster the development of extension points with enough "reuse-appeal" by those producers.

Overall, we are inclined to support the concerns formulated by Nierstrasz back in 1992: the plug-ins in our sample are reused, but as a complete sub-system that is built on top of Eclipse. The plug-ins which get to be extended by a third party are still, in their vast majority, the ones which are part of the basic platform. So, after all these years, it seems component producers still have trouble with the four main difficulties identified by Nierstrasz (as discussed in section 7.1).

7.7.3 Future work

This work can be extended in several ways, both by keeping the context to Eclipse plug-ins and by changing that context to other component models.

In the realm of Eclipse plug-ins, both close and differentiated replications would be useful. Close replicas of this study could use the same design, but a different sample of plug-ins (either from the same broker, or from a different one). Differentiated replicas could explore information not used here, such as the domain of application, and license type of the plug-ins, as well as the complexity of the extension points, just to enumerate a few examples. For instance, the complexity of an interface (in this case, the extension point) is often associated with the reusability of the component providing that interface. As such, it would be interesting to assess the impact of such complexity on the actual usage of the extension points mechanism.

The extent to which the conclusions in this study are applicable to components

from other component models should be addressed by further research. With some adaptations, our design can be adopted in other contexts.

The main adaptations would be to port the concepts of extension to a metamodel of the new component model, and build the data collection tools to analyze other component repositories. Naturally, it would also be necessary to identify the component frameworks that would play the equivalent "role" of the Eclipse standard distribution, in this new context.

Another interesting variation, if several competing (or complementary) component frameworks could be found, would be to change the design to a quantified multiple control group post-test only design, where each group of components would correspond to one of those frameworks and the remaining components would be considered as part of a special group.

Chapter 8

Conclusions

Contents

Background: *Throughout this dissertation we presented a process for conducting experimental work in Software Engineering, along with an approach called Ontology-Driven Measurement (ODM) to define and collect quantitative metrics that can be used in that experimental work. We illustrated the process, and ODM, with several examples of experimentation in the context of CBD.*

Objectives: *In this chapter our goals are to summarize our contributions toward achieving auditability and replicability in Experimental Software Engineering (ESE) applied to CBD, and to discuss possible extensions to our work.*

Methods: *We revisit the main problems outlined in the introduction of this dissertation, and discuss how our work contributes to solve them.*

Results: *The experimental process model and ODM are valuable tools that were successfully applied in quantitative experimental work in the context of CBD.*

Limitations: *The experimental validation of claims is never complete without the identification of the validity threats to that validation. By identifying such threats we can outline future work that complements our experimental work. In addition, we also identify future work that extends our contributions, both in what concerns the experimental process and ODM.*

Conclusions: *Experimental validation of CBD claims can be facilitated by an ESE process model, along with the usage of ODM. Together, they contribute to replicable experimental work that can be reconciled with the work of our peers to allow for meta-analysis that, in turn, facilitates advances in the body of knowledge of CBD.*

8.1 Summary

In this dissertation we have shown that we can facilitate the repeatability and comparability of quantitative experiments on component based development (CBD), by combining a well defined Experimental Software Engineering (ESE) process model with a formal approach to software measurement called Ontology-Driven Measurement (ODM). The repeatability and comparability of experimental work are two key characteristics of the experimental validation of claims, whether we consider science, in general, or ESE applied to CBD, in particular.

Repeatability is important so that the experiments can be replicated by independent researchers. This is a desirable property of experimental work, because it facilitates the independent validation of claims. By replicating experimental results reported by our peers (and facilitating their replication of our own work) we can systematically mitigate the threats to validity that are intrinsic to the experimental validation of claims.

The comparability of experimental results is essential, if we are to perform any meaningful meta-analysis of the results reported by independent researchers and practitioners on a particular subject. This comparative analysis of experimental results that support, or contradict, claims in the scope of CBD is a basic element for incrementally building the body of knowledge in CBD.

As we have seen in chapter 2, most of the work in Component-Based Software Engineering (CBSE), the area of Software Engineering dedicated to CBD, has been devoted to the development of component models and technologies, while the experimental validation of claims was relatively unexplored. An analysis to existing quantitative metrics for CBD revealed several metrics sets. These sets have problems concerning the coverage of concepts (the majority is dedicated to variations on the theme *structural complexity* of component's interfaces), the lack of connection to well-defined quality models, the lack of validation of metrics, and the ill-definition of those metrics.

In order to improve the state of the art in experimental quantitative approaches to support CBD, we followed a 2-pronged strategy, that covers both the process and the technical difficulties identified earlier.

Chapter 3 was dedicated to the process part of our approach. We were concerned with the tacit knowledge problem in experimentation. This problem relates to the amount of valuable information that is usually absent from experimental reports, severely hampering our ability to replicate experiments and perform meta-analysis on their results. We identified a set of disperse guidelines for conducting experimental work, and combined them into a model for the experimental process in the scope of Software Engineering. Our model was designed to complement existing proposals concerning experimental reporting guidelines. We also conducted a case study to validate the process model.

Chapter 4 was dedicated to the technical challenges in defining software metrics for

CBD. We used the ODM approach to formally define metrics for CBD. ODM combines the usage of an ontology, to describe the measurement targets, with OCL rules, to define and compute the metrics. ODM was illustrated in a cross-validation case study for a well-known CBD metrics set [Washizaki 03], in chapter 4, and exercised with two additional metamodels (UML 2.0 and CCM 3.0) and several other metrics sets, in chapter 5. The metrics formalization in chapter 5 covers several of the CBD metrics sets identified in the related work, to show the praticality and expressiveness of ODM.

Chapters 6 and 7 are dedicated to the validation of our claims concerning the usefulness of the combination of our process model, along with the ODM approach, in the context of CBD. They provide two examples of how our quantitative experimental approaches can provide us insights on CBD.

Chapter 6 contains a case study on the development process of software components. We focus on one particular activity during component development: code inspections. Our experimental work explores the impact of practitioner's expertise in the outcome of software code inspections. The code inspections were carried out in the context of a software components factory, simulated in an academic environment.

Chapter 7 contains an observational study on the reuse patterns within a well known component-based software system: the Eclipse project. We explored a different angle of component reusability, when compared to the one used in the vast majority of metrics-based approaches to assess component reuse. Rather than using existing metrics on the component interfaces, and their complexity, and using those metrics to indirectly compute the component's reusability, we defined and used metrics for the actual reuse of components. In this study, we used a particular kind of software components: Eclipse plug-ins. Our metrics treat Eclipse plug-ins as black-box components, even when those plug-ins are open source. There are two main motives for this: (i) frequently, components are distributed as black-boxes, so our experiment can be replicated for other kinds of components, regardless of the visibility level of their features; (ii) one of the main benefits attributed to CBD is to foster the reuse of components which can range from fine- to coarse-grained components; even when it is possible to access components as white-box, it is often undesirable, or unfeasible with the available resources to do so.

Our observational study indicated a reuse pattern which is independent from the structural properties normally assessed by other metrics-based approaches to CBD: the credibility of the component provider seems to be a dominating factor, when selecting software components for reuse, particularly if we are to build components that reuse the third party components. While practitioners download and install components from several sources, they are much more conservative when it comes to developing new components which depend on such third-party ones.

It seems plausible that component producers are not willing to make the usage of their components dependent on third-party components, because the component mar-

ket is not mature enough so that they are viewed as commodities that can be easily replaced, if necessary, by alternatives produced by other vendors. The lack of such alternatives makes the evolution of the third party components a risk. It is difficult to guarantee that new versions of those components can safely replace earlier versions, in a given component assembly, with the current component models. This risk is mitigated in well-known and often reused component frameworks (e.g. the Eclipse platform plug-ins). Breaking backward compatibility is a more serious risk for those framework's producers, due to the large number of clients which depend on the framework's features. One of the alternatives for the framework's producers is to adopt a deprecation policy, which keeps old components available, while encouraging their replacement by new ones.

8.2 Contributions

The most significant overall contribution of this dissertation is

the proposal and validation of an experimental process model combined with with Ontology Driven Measurement to support the experimental validation of claims in the context of CBD, in a repeatable and comparable way.

In this section, we outline the main contributions of this dissertation that, when combined, lead to this overall contribution.

8.2.1 Metamodels construction and extension

The availability of a well-defined ontology for the targets of our experimental work is a concern that cross-cuts the whole dissertation. We consider this as one of the essential elements to support replicability and comparability of results.

As such, throughout the dissertation, we modeled the domain upon which each of our experiments were carried out. In several cases we created new ontologies, or extended existing ones.

The exception is in chapter 4, where we used an extract of the standard UML 2.0 metamodel, with no further changes. The standard metamodel was suitable for our purposes.

UML 2.0 metamodel extension

In chapter 5, we extended the UML 2.0 metamodel by adding a stereotype for representing events in UML 2.0 component diagrams. The stereotype allows representing provided and required events, similarly to what is possible with provided and required interfaces, thus adding to the expressiveness of UML 2.0.

CCM 3.0 metamodel extension

In chapter 5, we discussed how the standard CCM 3.0 metamodel allows representing models of component assemblies, but lacks the expressiveness required for specifying instances of component assemblies. In other words, it is possible to know the types of components which may be wired, but not which specific components are actually wired. In CCM 3.0 this wiring can be specified through a textual file called Component Assembly Descriptor (CAD). This contrasts with other metamodels, such as the one of UML 2.0, which provide the abstractions required for representing models and instances. The extension described in chapter 5 solves this problem by adding the meta-classes that are required to represent the contents of the CAD files through the extended metamodel.

Code inspections ontology

In chapter 6, we modeled the process under scrutiny in our case study: code inspections. The resulting ontology also includes the concepts required for representing the expertise of the participants in our case study. This is an example of an ontology created from scratch to support the ODM approach. Note that, although this ontology is not defined at the metamodel level, the ODM approach applies to it in a similar way. This illustrates the meta-level independence of ODM.

Eclipse plug-ins metamodel

In chapter 7, we developed a metamodel to represent Eclipse plug-ins, their dependencies, and meta-information provided by a plug-ins broker about the products in which the plug-ins are bundled. The metamodel captures the information conveyed by Eclipse plug-ins manifest files, and adds to that information extra data provided by an Eclipse plug-ins broker. In this metamodel, plug-ins are represented as black-box components. It excludes white-box information on those plug-ins, even when that information is available (e.g. in open-source plug-ins), because, for the purposes of this dissertation, we were only interested in the black-box view of the plug-ins.

UML profile for Acme

In appendix B, we present a UML 2.0 profile for specifying Acme components and systems. Acme is a second generation Architecture Description Language (ADL) designed to be used as an interchange language between different ADLs. By creating a mapping between UML 2.0 and Acme, we are indirectly creating a mapping between UML and those ADLs for which there is a mapping to Acme. We do not use this profile in any of the experimental works described in this dissertation, because we chose to use a more widely disseminated component technology (Eclipse plug-ins) in our experiments. Nevertheless, we include our UML profile for Acme in this dissertation, as

it may contribute for future experimental work with components defined in Acme, or any of the ADLs that map to Acme. The rationale is to use the mapping to represent such components, and component assemblies into our profile, and then use the ODM approach to define and compute relevant metrics on them.

8.2.2 Quality models and their validation

As discussed in the introduction, we opted to use focused quality models, or small sets of quality attributes, in our experimental work, rather than using a generic quality model for CBD. Our contribution concerning quality models was focused on a quality model proposed by Washizaki *et al.* [Washizaki 03], for which we performed a cross-validation, in chapter 4.

In chapter 4's case study, our contribution concerns only the external validation of that quality model. We used a set of fine-grained components, developed by Washizaki *et al.*'s team for educational purposes (in particular, as examples of reusable components). The components turned out not to conform to some of the heuristics proposed in the context of the quality model. Our results suggest that the quality model's heuristics may be vulnerable to component size, because the components used for calibrating the heuristics were commercial components, and are likely to have a higher complexity than those used in our case study. In summary, the quality model may lack external consistency.

The context for other experimental works was provided by small sets of quality characteristics, rather than by a full-blown quality model.

8.2.3 Formalization of metrics for CBD

Throughout this dissertation we presented formal specifications for metrics that cover not only the product, but also the process. These formal specifications are built using the ODM approach. The ODM approach is an extension of the Metamodel-Driven Measurement (M2DM) approach to allow measurement in meta-levels other than M2. M2DM was originally created to support the specification of metrics for object-oriented design. While extending the usage of M2DM to CBD, we found that it would be useful to define and collect metrics at other meta-levels, rather than being restricted to the metamodel level. The ODM evolution was used with new metamodels, such as the CCM metamodel, or our Eclipse plug-ins metamodel, but also the usage of different parts of other metamodels which had been previously used with metrics for object-oriented design, such as the UML 2.0 metamodel. We also extended the ODM approach to process evaluation, namely in the case study presented in chapter 6, on the effect of expertise in inspection teams. The variety of metrics defined using the ODM approach, throughout the dissertation, illustrates the expressiveness of the approach.

Assembly-independent component metrics

Assembly-independent component metrics are metrics for software components that can be computed without any extra information concerning the assemblies in which components are integrated. In chapter 4, we formalized the definitions of a metrics set by Washizaki *et al.* [Washizaki 03] for *JavaBeans*, so that we could cross-validate the metrics set. The formalization was performed upon the UML 2.0 metamodel. In chapter 5, we formalized metrics originally proposed by several authors [Boxall 04, Narasimhan 04, Goulão 05b] upon the UML 2.0 and the CCM 3.0 metamodels. As discussed in section 8.2.1, some of these formalizations required extending the standard metamodels.

Assembly-dependent component metrics

Assembly-dependent component metrics are metrics for components, or component assemblies, which require information concerning the component assemblies so that their value can be computed. Unlike assembly-independent metrics, these metrics' values are not simply a characterization of components in isolation, but rather a characterization of the relationships among components, or of the assembly as a whole. In chapter 5 we formally defined several metrics proposed in literature [Hoek 03, Narasimhan 04] upon UML 2.0 and CCM 3.0. In chapter 7 we defined and validated metrics for Eclipse plug-ins upon an Eclipse plug-ins metamodel designed in the context of this dissertation. This metrics set, also proposed in the context of this dissertation, is dedicated to the usage of the extension mechanism of Eclipse plug-ins. It allows contrasting the potential usage of this mechanism with its actual usage.

Process metrics (for code inspections)

In order to exemplify how metrics can be defined in the context of the CBD process, we focused on one of the activities that can be carried out in that process: code inspections carried out by the component producer organizations. The metrics were defined in the context of this dissertation, and cover the diversity of defects found during code inspections, as well as the individual expertise of the members of the inspection team. We used the combination of these two dimensions (inspection success and individual expertise of team members) to explore how the inspection team composition might be optimized.

8.2.4 Validation of proposals through a common process model

All the experimental validations performed during this dissertation followed a common process model, described in chapter 3. In that sense, each of those experimental validations can be viewed as an example of the application of the process itself. This

validation effort in chapter 3 was aimed at ESE in general, rather than to specific applications of ESE to CBD. The remaining experimental works described in the dissertation focused on CBD.

Validation of the process model

In chapter 3 we presented a case study where we assessed the relative difficulty of performing each part of the process. The major difficulties of our subjects in following the process concerned data collection, although it is unclear whether their lower performance while describing the data collection activity was intrinsic to the process, or a result of the characteristics of the experimental setting.

Cross-validation of a component metrics set and quality model

In chapter 4 we performed a cross-validation of Washizaki *et al.*'s metrics and quality model [Washizaki 03] and found problems with the external validity of the quality model. When assessing components created for educational purposes, rather than commercial ones, several of the quality thresholds defined by Washizaki *et al.* were not satisfied by most of the components in a set also provided by Washizaki's team. The smaller size of educational components, when compared to the commercial ones used while creating the thresholds is a likely cause for this lack of external validity.

Validation of inspection team's desirable properties

In chapter 6, we performed a case study on a sub-process in the context of the development process of software components: code inspections. The main outcome of this case study was that it is possible to optimize the inspection team so that we minimize the number of expert reviewers involved in the inspection, without sacrificing the inspection's outcome.

Validation of claims concerning the reusability of Eclipse plug-ins

In chapter 7, we performed an observational study on the reusability of Eclipse plug-ins. The main outcome of this study was the confirmation of the perennial nature of several observations of Nierstrasz *et al.* [Nierstrasz 92]. These include the difficulty in creating reusable components, and the prevalence of the reuse of components which are part of a trusted framework, rather than the reuse of components from a diversity of sources, acquired through a component broker.

8.2.5 Development of tool support for experimentation

We developed the tool support required for our experimental work in a systematic way. Our experimental process model, with several sub-processes performed in sequence,

can be mapped in a pipe and filter architecture, where the outputs of an activity are used as inputs to the next one. Our pipe and filter architecture provided support for the data collection and subsequent statistical analysis.

If we compare the tool support for each of the experimental validations, a family of architectures emerges. We use a custom component for collecting the data and transforming it into instances of an appropriate metamodel. The metamodel also changes from one experimental validation to the next. Then, we use this instantiation in the USE tool, so that we can collect metrics defined in OCL upon the metamodel. The results of the OCL queries are fed into a transformation tool that formats them for further analysis in our second off-the-shelf component: SPSS.

The option for a pipe and filter architecture allowed the usage of loosely coupled components. Each of them is a stand-alone application. This option favored the replaceability of the custom components, from one experiment to the next, as well as the independence of the custom components from the technology used in their development. The interoperability between components was always assured by well defined text file formats. These formats were fixed for the off-the-shelf components' inputs, as both selected components allow importing data through textual scripts. We defined convenient formats for the custom made component's inputs. Further details on the tool support created for our experimental work can be found in appendix C.

8.3 Future work

We organize the discussion of future work into two main streams: one dedicated to advancing the current state of practice in Experimental Software Engineering, and another dedicated to extensions on the experimental work in Component-Based Development that would complement the work performed in the scope of this dissertation.

8.3.1 Experimental process improvement

Systematic review of experimental design usage

The experimental process formalization work, described in chapter 3, was mostly based on surveys concerning the state of practice in Experimental Software Engineering, as well as tentative guidelines to leverage the benefits of the experimental work by streamlining the whole experimental process. A common recommendation found in the latter concerns the usage of standard experiment designs, rather than custom made ones, as this would help researchers and evaluators to assess the adequacy of the experimental protocol used.

Each design has its own strengths and weaknesses. The latter are related to common threats to validity that are inherent to them. To the best of our knowledge, there is no systematic attempt to classify the existing literature in Experimental Software En-

gineering with respect to the usage of experimental designs. Such a survey would be useful to the community in that it would highlight patterns of usage of experimental designs which could, in theory, prove useful to detect weak spots in the experimental validation efforts conducted by the community. If the experiments concerning a given problem share a common threat to validity that was not explicitly dealt with, being aware of such prevailing threat can help guiding future experiments, so that their chosen design can address it.

A systematic review of controlled experiments design usage would represent a step forward, when compared to current practice. It would help defining a road map for required experimental work, so that the combined set of experiments addressing a particular problem would minimize the remaining threats to validity. The overall goal would be to facilitate a more rational building of the body of knowledge concerning the problem addressed by a set of experiments. We consider two complementary variants of these reviews: a generic and a focused one.

The generic review could extend an existing review [Sjøberg 05] on controlled experiments by using the same collection of publications as in that paper (perhaps updating the collection with new publications on the same journals and conferences not considered in the original paper). The analysis could focus not only on the frequency of experimental designs, broken down by the Software Engineering sub-areas defined in the Software Engineering Body of Knowledge [Abran 04], but also on their evolution over time. This would allow the identification of patterns in the used experimental designs and their trends.

Focused reviews would cover a specific topic (e.g. code inspections) to facilitate the meta-analysis on existing work and results, as well as the identification of remaining validity threats that could be addressed by further experimentation.

A product line for experimentation

During our research activities within CITI and, more precisely, within the QUASAR [1] research group, we have noted that although we can identify commonalities among the experimental tool support developed in our work and that of our colleagues, there is little usage of synergies in tool support construction, other than the exchange of know-how in assembling those tools.

As we have discussed in section 8.2, we were able to build a family of pipe and filter architectures that provided the tool support for the experimental work described in this dissertation. We were also able to single out points of variation in these architectures, as well as their commonalities. Nevertheless, we built custom tool support for each of the experiments, even if this support follows a common architectural style and shares some of its components.

The challenge is to leverage the effort spent in tool support, so that we can facili-

[1] http://ctp.di.fct.unl.pt/QUASAR/

tate the design and execution of Software Engineering experiments. Software product lines provide a good basis for the kind of tool support we would like to build. Rather than the opportunistic reuse that we currently use, we should strive for designing a well-defined product line and build the next generation of components to support our experimentation activities into those product lines.

A Domain Specific Language for experimentation

The configuration of the experimental work in the previously mentioned experimentation product line should be performed using a Domain Specific Language (DSL) for experimentation. The DSL should provide adequate constructs to specify the experimental process, from requirements to results packaging. The idea would be to integrate several of the elements used in this dissertation, to facilitate experimentation. We would still use an ontology based approach to represent the experimental data. The DSL would support the experiment's design planning, so that all the hypotheses and used variables would be expressible in terms of that ontology. Furthermore, the DSL would also support the selection of an experimental design, the specification of the collection process, analysis techniques, and instrumentation.

The specification of a DSL for experimentation would allow encoding best practices rules into the DSL. To illustrate this, consider the following examples:

- The identification of independent and dependent variables, as well as their scale types could automatically narrow down the selection of available statistical techniques that could be applied in this situation.

- Upon the selection of a particular experimental design, the appropriate set of threats to validity that result from that design choice could be automatically selected and included in a template report of the experiment.

- It is common to build well-formedness rules into DSLs, so that the designs modeled with those languages can be automatically validated against those rules. Therefore, building rules based on experimentation best practices into a DSL for experimentation and automatically checking those rules would facilitate the conduction of sound experimental work.

Last, but not the least, such a DSL would contribute to mitigate the tacit knowledge problem described in chapter 3. In fact, along with the underlying tool support, it would also facilitate the replication of experiments.

8.3.2 Extensions to our experimental work

In this dissertation we have provided several examples of how to combine the process model described in chapter 3 with ODM, to support quantitative experimentation in

CBD. We sampled these techniques with a variety of component models, as well as by using them not only in the course of experiments concerning the product, but also the development process. Although we have strived for variety in these examples, we are far from exhausting the plethora of potential applications of these techniques.

Cross-validation of proposals

As noted in section 2.5.7, most of the metrics proposed for CBD (including the ones proposed by us, in chapters 6 and 7) would benefit from external validation efforts. In this dissertation, we conducted a cross-validation on one of the existing metrics sets. It is possible to extend this validation with replications using other experimental designs and samples. It would also be interesting to build up on our formalization effort in chapter 5, and use those formalized metrics in external validation efforts for each of them.

Process experiments

The case study in chapter 6 can be extended in several ways, namely through differentiated replicas, where we vary not only the subjects, but also several features of the experimental design, to mitigate the threats of validity identified in that chapter. Among other options, we would try alternative ways of determining the expertise of practitioners involved in the experiment, such as using a pre-test to assess their skils as reviewers, rather than our indirect measures of expertise. Another evolution would be to compare team's performance with that of an independent and thorough assessment of the inspected components, to allow for an alternative assessment of the success of the inspections.

We would also like to extend our study on the effect of individual's skills in software development teams to other sub-processes of that development, as this would be useful for project managers.

Product experiments

We would like to extend the study in chapter 7 to a significantly larger sample, to confirm the external validity of our findings in the population of Eclipse plug-ins.

With some adaptations, we could also conduct similar experiments with components built according to other component models. This would allow assessing the external validity of our findings to such models. If the observed discrepancies among the reuse patterns of plug-ins in the basic platform *vs.* those found in the plug-ins brokers are generic to other component models, this information would be useful for understanding reuse patterns in those models, as well. If, in contrast, we were to find out significantly different patterns of reuse, more in line with the long sought "El Dorado" of a component market, understanding the fundamental differences between

such component models and the Eclipse plug-ins model would allow a deeper understanding of what would make a software component market work. We believe this would be an important step forward in the identification of such component market mechanisms, from educated guesses based on an individual's experience, valuable as these may be, to quantitatively supported evidence.

Time series analysis

A time series is a collection of observations made sequentially in time [Chatfield 84]. These observations may be continuous, or discrete (the latter are typically equally spaced observations). Time series analysis has four main objectives: to describe, explain, predict, or control the evolution of those observations.

The description can be used to detect trends, seasonal effects, and observations that do not seem consistent with the rest of the data, which can correspond to external events, and even turning points in the time series. When we observe the evolution of more than one variable, we may try to explain the evolution of a variable through the evolution of other variables. We may also try to predict the evolution of a time series, given the observations in that series in the past. Finally, we can also use time series to control the evolution of a process. All these objectives can be useful in a quantitative analysis of the evolution of a component-based system, or a component framework, for instance, to support planning the evolution of those systems, or frameworks.

Although we have not used it in this dissertation, we plan to extend our work by performing time series analysis in the development of open source component-based products. The first candidate for analysis is the Eclipse platform itself. The advantage of using an open source repository for this kind of observational study is that we can access not only the complete evolution tracking system repository and extract our observations from such repositories, but also that we can access the project's version system. This way, we can match the bug tracking reports with the corresponding changes in the software components that make up the whole project.

This sort of analysis would increase our understanding on the evolution of component-based systems, and of quality attributes such as their maintainability and stability, over time, as well as the effects of development milestones and releases in those characteristics.

The uniformity of technologies (Eclipse plug-ins implemented in Java) used in this component-based system, its continuous evolution, and the easy access to its repositories are some of the main features that lead us to choose this system as our target for evaluation. In time, we hope to extend this study to other projects (e.g. Mozilla).

[This page was intentionally left blank]

Appendix A

Component models

Contents

Background: *In chapter 2, we briefly discussed several component models and some of their main features.*

Objectives: *In this appendix, our goal is to provide a more detailed overview of those component models.*

Methods: *We start by presenting a toy example that will be used to illustrate some of the features of the component models. Then, for each component model, we discuss the model's origin, the component's representation, syntax, and integration kinds, the model's structure, and the support provided by the model to contracts, certification, and compositional reasoning.*

Results: *This systematic comparison of component models allows an overall perspective of the state of the art in component technology.*

Limitations: *The inclusion and comparison criteria used in this survey are aimed at a representative set of current component models and of their main features. A more exhaustive survey would be possible, but out of the scope of this dissertation.*

Conclusions: *We can observe a great variety of the approaches of each component model, according to most of the used criteria. Perhaps the most notable commonalities concern the generally low level of contract support, the very limited support for compositional reasoning and the non-existing support for the certification of components.*

A.1 Introduction

A **component model** is *"the set of component types, their interfaces, and, additionally, a specification of the allowable patterns of interaction among component types. A **component framework** provides a set of runtime services to support and enforce the component model"* [Bachman 00].

In this appendix, we will briefly discuss some of the main component models used in industry and academia. We start describing a toy example representing a component assembly for a clock system. Then, using the taxonomy presented in chapter 2, we will briefly discuss several existing component models. The models will be illustrated by using the clock system example. Note that the level of details provided for each of the surveyed component models may vary, when representing the clock system as our purpose is only to provide an intuition on how the components are represented in each component model, rather than providing a complete specification of our toy example for any of them.

A.2 A toy example

The clock system example consists of a component assembly that integrates two components: a clock, and a display.

The clock is characterized by the following properties:

- hours - an integer value, such that $0 \leq \text{hours} \leq 23$
- minutes - an integer value, such that $0 \leq \text{minutes} \leq 59$
- seconds - an integer value, such that $0 \leq \text{seconds} \leq 59$

At regular intervals (e.g. every second), the clock sends an event called tick, indicating that it has updated the current time. The clock also provides information concerning the current time.

In our example, the clock is integrated with a display component. The display is characterized by the following properties:

- horizontal - an integer value, with the horizontal size of the display
- vertical - an integer value, with the vertical size of the display
- minimum refresh rate - a real value, indicating the minimum time interval elapsed between two consecutive refreshes of the display that this component can support.

The display component listens to the tick event fired by the clock and can ask the clock for the current time, so that the display information can be updated.

A.3 Inclusion criteria

The following sections briefly present a set of component models. With respect to the inclusion criteria for the presented component models, we chose models with a high visibility in the community. These include models commonly chosen in surveys such as [Lau 07], or presented as typical component models examples in books such as [Heineman 01, Szyperski 02, Crnkovic 02].

A.4 Component models

A.4.1 JavaBeans

Origin [industry].
Sun proposed the JavaBeans component model, with the purpose of allowing third party independent software vendors to create and ship components that could be composed together into applications by end users [Hamilton 97]. It became a popular component model, partly due to its relative simplicity, when compared to more sophisticated component models, such as the CCM [Estublier 02].

Although the JavaBeans component model was designed mostly for creating graphical user interfaces (GUIs), its scope of application is not limited to that domain. It is also possible to create "invisible beans" and compose them into applications (with or without a GUI). JavaBeans may range from fine-grained building blocks (e.g. a button) of an application to application-like components (e.g. a spreadsheet to be embedded in a web page). In other words, most components are small, although medium sized components may also be defined.

Component representation [class].
A JavaBean is a special Java class. The JavaBean class may wrap an existing object, or set of cooperating objects, as well as call objects outside the boundaries of the JavaBean component.

Component syntax [object-like programming language].
JavaBeans may be described through an interface that is essentially a collection of **features** (which can also be described as ports). These features can be of several types: **methods**, **properties**, **event sources**, and **listeners** (event sinks). Figure A.1 illustrates the available interface of a JavaBean.

Properties are used both to parameterize the components during their integration and as attributes, in the object-oriented sense of the term, during run-time.

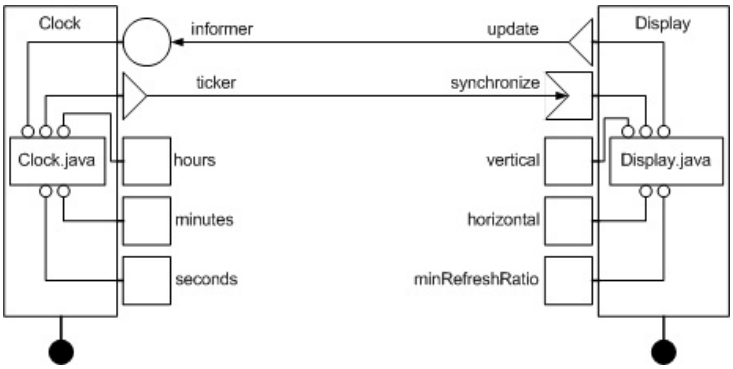


Figure A.1: JavaBean's interface features

Events are a mechanism to allow components to be plugged together in an application builder. Some components will act as event sources while others act as event listeners. The former generate events that are caught and processed by the latter, or scripting environments [Hamilton 97]. In our clock system example, we can see how each component is responsible for starting an event. The `Display` component has a listener that captures the event sent by the event source `ticker`. The `Clock` component provides a method `informer` that is called whenever the `Display` component starts the `update` event.

JavaBeans's methods are the only operations associated with the component's features. In other words, the JavaBean's properties (e.g. `hours`, `minutes`, and `seconds`, in the `Clock` component) are local to the JavaBean, and can only be accessed through the JavaBean's operations. If we want to characterize the features as "required" and "provided" features, external source events can be viewed as required features, whereas methods and listeners can be regarded as provided features. The "black lollipop" below each component represents the respective component' whole interface.

Component integration [deployment with repository].
JavaBeans may be developed using a common Java development environment, but are then stored in the toolbox of the Bean Development Kit (BDK), which is a component repository. To this purpose, JavaBeans have to be packaged into an archive with the resources they depend on, which may include images, configuration files, and other Java classes. It is also common to package together JavaBeans that share several resources, to avoid resources duplication in the repository [Estublier 02]. At runtime, the bean can be retrieved from the repository.

Model structure [flat].
The component model is flat.

Contract support [basic syntax/type contract].
JavaBeans provides no direct support for design by contract, other than the type

system of the Java language.

Support for certification [not available].
JavaBeans provides no special support for the certification of components.

Support for compositional reasoning [not available].
JavaBeans provides no special support for compositional reasoning.

A.4.2 Enterprise JavaBeans

Origin [industry].
The Enterprise JavaBeans (EJB) component model was originally developed in 1997 and has evolved since then to its current version (EJB 3.0 [DeMichiel 06]). It is aimed at server-side components that encapsulate the business logic of applications. EJB provides standard support for concerns such as persistence, transaction processing, concurrency control, events, naming and directory services, security, distribution, and deployment of the software components in an application server.

The fact that EJB is a distributed component model may impact design decisions concerning the granularity of components, because using fine-grained components, which are supported by the model, may lead to an inefficient implementation due to the network traffic involved in the communication between the fine grained EJBs. Instead, it has been argued that it is better to develop coarser-grained components that consist of multiple Java classes, in order to create reusable, self-contained, business components, than to follow a design approach closer to traditional OO design, with each component corresponding to an object [Lublinsky 04].

Component representation [class].
In its most simple form an EJB is a Java class, hosted and managed by an EJB container which is provided by a J2EE[1] server. In practice, rather than just a class, the EJB can be implemented using several cooperating classes, with one of them acting as the bean class. The EJB container supports security, transaction management, and several other services, as discussed in the *Origin* sub-section. In particular, J2EE supports remote connectivity between clients and EJBs.

Component syntax [object-like programming language].
An EJB component is specified through the **bean class**, along with a **home** and a **remote interface**, if the bean is an entity, or represents a session. The bean class implements the functionalities of the EJB, possibly in cooperation with a set of other classes. The home interface defines the operations available for the EJB, from the point of view of the EJB's life cycle. This includes creation, location and destruction operations for

[1] **J2EE** - **J**ava **2** **E**nterprise **E**dition

the EJB. The remote interface defines the operations made available by the EJB for its clients. EJB does not explicitly support the specification of required interfaces.

EJBs can be of the following types:

- **Entity beans** are used for modeling business data, along with the operations to manipulate it. This business data is stored persistently in a database. Their persistence can be managed either by the container or by the bean itself.
- **Session beans** are used for modeling business processes. As such, these beans do not keep their data persistently. Nevertheless they can keep state, if the state of the communications with the EJB's clients is required to implement their interaction protocol. In any case, this state is not stored persistently.
- **Message-driven beans** are used for modeling message driven business processes. They act as Java Message Service listeners and cannot be accessed through an interface, in contrast with session beans.

Our clock system running example is not a typical application for the EJB component model. EJB is targeted for server side components, so the display component would normally be a client application, rather than an EJB. The clock component could be implemented in several different ways, and in most situations we would probably not model it as an EJB, as well. However, just for the sake of presentation, let us assume that we do want to represent the clock component as a stateless session bean. In figure A.2 we represent the `Clock` session bean, with its home interface (`ClockHome`), where life cycle methods are declared, and its remote interface (`Clock`), where the business methods are declared. The `ClockBean` class implements the bean's code. If necessary, we could define helper classes for `ClockBean`. Note that, along with the already known properties for storing the time, we would add an extra property to store context information (`ctx`), as well as some methods required by the EJB container for supporting a stateless session bean, none of which are directly called by the EJB's client (`ejbCreate`, `setSessionContext`, `ejbActivate`, `ejbPassivate`, and `ejbRemove`). The details of the attributes and operations are omitted from the class diagram, for simplicity.

In a nutshell, the EJB developer has to implement a set of Java classes and interfaces, adhering to a set of predefined conventions, so that his classes can be used within the scope of the EJB component model.

Component integration [design with deposit-only repository].
The beans are stored as a JAR archive, along with an XML deployment descriptor where the security, persistence and transactions specifications are recorded. At design time, the enterprise beans are integrated into an EJB container. If a bean X requires a service provided by bean Y, then Y has to be stored into the EJB container before X. However, beans X and Y remain as two different beans in the EJB container, rather than as an integrated bean.

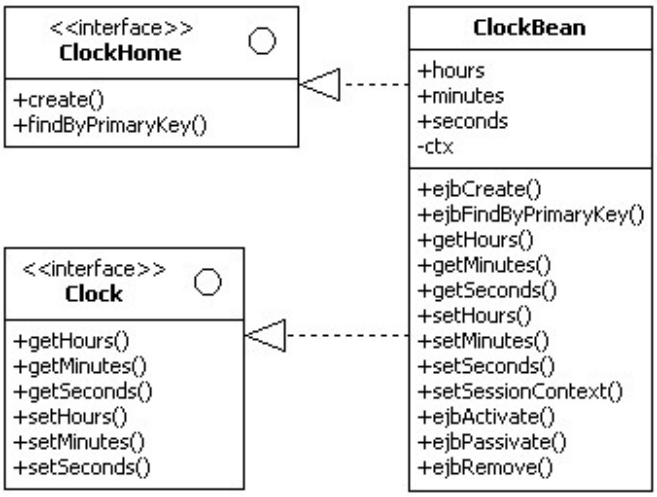


Figure A.2: Enterprise JavaBean's example

The beans stored in the EJB container are then made available by the J2EE server to external clients (in our example, through the ClockHome and Clock interfaces, which are exposed by the EJB container), but the abstraction of integrated bean built in the design phase of the EJBs is not available at the deployment time.

Model structure [flat].
From the above discussion on component integration, we can infer that the component model is flat.

Contract support [basic/syntax-type contracts].
EJB provides no direct support for design by contract, other than the "basic syntax/-type contract".

Support for certification [not available].
EJB provides no direct support for independent certification.

Support for compositional reasoning [not available].
EJB provides no direct support for compositional reasoning.

A.4.3 COM+

Origin [industry].
COM was created by Microsoft in 1995 as an approach to achieve program independence (by using components) and languages independence, in a centralized context, on Windows platforms [Estublier 02]. It uses interfaces and binary interoperability conventions to support interaction among components. The model was extended to

DCOM, with the support for distribution, and later to COM+[2], with the inclusion of support for persistence and transaction services.

Most COM+ components are coarse-grained. Indeed, COM+ components can be used by client applications and behave, from those application's point of view, as a complete system [Lau 05a].

Component representation [object].
Components are binary objects. The COM+ component model is strictly a run-time model.

Component syntax [programming languages with IDL mappings].
COM+ interfaces are similar to C++ virtual classes, with a list of methods and attributes, but no implementation. Figure A.3 presents the `Clock` and `Display` COM+ components. Each component has a client relationship with the other one. In this example, we encapsulate the methods provided by Clock in the interface `IClock`, and the methods provided by `Display` in the interface `IDisplay`.

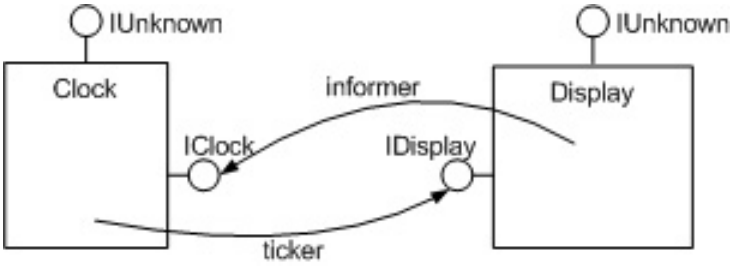


Figure A.3: COM+ components

Although interfaces are defined using a language independent **I**nterface **D**escription **L**anguage (**IDL**), developers write their code in a COM+ enabled language, such as C, or C++. The IDL representation can be automatically generated, so developers do not need to use it directly. All COM+ components implement the `IUnknown` interface, which defines a basic set of operations all COM+ components need to support, such as query methods that allow clients to dynamically discover which interfaces each component supports.

Component integration [design with deposit-only repository].
COM+ has no integration language. Instead, it relies on a protocol to allow COM+ components to register into a component registry, and lookup available components with which they can interact, as well as creating instances of those components. In practice, this is achieved by making all COM+ interfaces descendants of the `IUnknown` interface, which contains the basic functions.

[2] `http://www.microsoft.com/com/`

Rather than interacting directly with other components, clients use interface pointers to access other components. Integration is achieved through methods calls made from one component to an interface pointer of another component. These interactions are coded at design time.

COM+ supports two integration mechanisms: containment and aggregation. Containment allows a COM+ component to be built by integrating another COM+ component in it, in such a way that the outer component's clients cannot see the inner component. The outer component may declare some of the inner component's interfaces and then delegate the calls to those interfaces to the inner component, but that is not visible outside the outer component. Aggregation allows the outer component to expose interfaces of the inner component as if the outer component implemented them, but does not need to implement these interfaces, as in the containment. This requires the source code of both the inner and outer components to be changed, to avoid conflicts with respect to the interface `IUnknown`.

Model structure [hierarchical].
The containment and aggregation mechanisms, described in the previous section, supports a hierarchical structure for COM+ components.

Contract support [basic syntax/type contract].
COM+ provides no direct support for design by contract, other than the "basic syntax/type contract".

Support for certification [not available].
COM+ provides no direct support for certification of components, or assemblies.

Support for compositional reasoning [not available].
COM+ provides no direct support for compositional reasoning.

A.4.4 .Net

Origin [industry].
.Net [3] is Microsoft's current component model. It is an evolution from previous component models from Microsoft as it no longer constrained to binary interoperability. Instead, the .Net platform provides **C**ommon **L**anguage **I**nfrastructure (**CLI**) support. CLI can be broken down into a **C**ommon **I**ntermediate **L**anguage (**CIL**), and a **C**ommon **L**anguage **R**untime(CLR). .Net languages are compiled into CIL, which is a platform neutral language, comparable to Java bytecode. CLR is a virtual machine, comparable to the Java Virtual Machine, which compiles CIL to machine readable

[3] `http://www.microsoft.com/net`

code, for execution. .Net supports from fine to coarse-grained components.

Component representation [object].
A .Net component instance can be viewed as an object, with a set of provided and required interfaces and events. It has a set of modules, which are executable files, or dynamic link libraries (DLLs).

Component syntax [programming languages with IDL mappings].
Components are developed in a programming language, such as C♯, C++, or Visual Basic, among several others. The compiler is responsible for transforming the component's source code into CIL, which runs on top of CLR. The compiler also generates a component descriptor, known as its **manifest**. The manifest contains meta-information about the component, including its resources, meta-data, code, and the lists of exported and imported events and methods. In .Net, the interface description of a component is called **assembly**. This description is flexible, in the sense that it is possible to define **custom attributes**. These can be used to implement service contracts, for instance, and are interpreted by the application code, rather than the CLR.

Component integration [design with deposit-only repository].
During compilation, the manifest is generated, along with the main executable file and auxiliary DLLs. Components cannot be made hierarchical, because the modules used in a component cannot be components themselves.

.Net supports non-functional properties such as distribution and security as part of the operating system and component dynamic loader. It has a visibility control mechanism which allows components to be local to an application, so it is possible for several instances of the same component and its constituents (e.g. DLLs) to be used simultaneously. This implies the usage of a version control mechanism, to support the dynamic loading of executables and DLLs, so that each component gets to interact with the correct versions of other components and of its own modules.

Component wiring is specified at development time, either in the source code, or while building the manifest. The correct versions of executables and DLLs are selected according to a set of rules expressed in XML. Although there is a default for those rules, they can be tailored, both system-wide and for a particular application.

Model structure [flat].
As explained in the beginning of the previous section, the .Net component model is flat.

Contract support [behavior contracts].
.Net provides support for several kinds of contracts. The syntactic contracts use the usual IDL features, with the explicit publication of provided and required interfaces, and operation signatures. Behavior contracts, with pre and post conditions, as well as invariants, are supported in recent .Net languages (namely, in Spec♯). The custom attributes defined in the assembly can be used for implementing extra levels of contract support. In particular, they can been used to support the definition of quality of service contracts.

Support for certification [not available].
.Net provides no direct support for the certification of software components.

Support for compositional reasoning [not available].
.Net provides no direct support for compositional reasoning.

A.4.5 CCM

Origin [industry].
The Corba Component Model (CCM) [OMG 02a] is the OMG standard for the specification of software components. As such, it is independent from a specific vendor, both in what concerns the component's programming languages and platforms. CCM components can range from fine to coarse-grained, but tend to be coarse-grained.

Component representation [object].
CCM components are server-side components, installed into a container, which is responsible for running them in the application server. The clients of a component can access the corresponding container through a naming server which is used for either creating or obtaining a reference to the component.

Component syntax [programming languages with IDL mappings].
CORBA components are created and managed by homes. A **home** is a meta-type which offers standard factory and finder operations and is used to manage a component instance's life cycle, including its creation, retrieval, and destruction. Components run in **containers** that handle system services transparently and are hosted by generic application component servers (e.g. OpenCCM [4]). Each component may have several **provided** and **required interfaces** (also known as **facets** and **receptacles**, respectively), and has the ability to **publish** and **subscribe** events (by means of **event sources** and **sinks**). Components also offer navigation and introspection capabilities.

Figure A.4 presents the clock system example, where several of these elements are visible.

[4] http://openccm.objectweb.org/

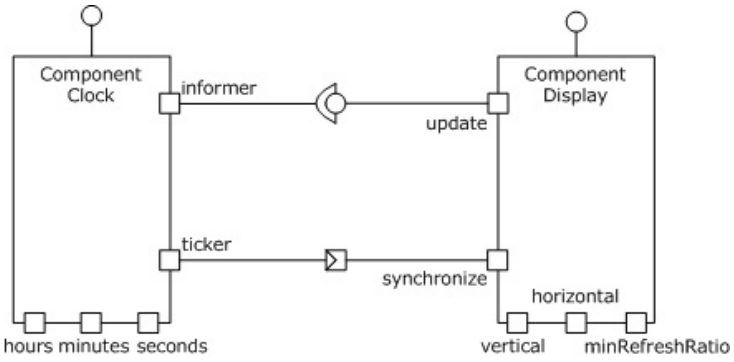


Figure A.4: CCM components

CCM components can be specified using OMG IDL 3. In listing A.1, some details are omitted, for simplicity. Note that this specification defines component types, but not how the component instances are to be wired. For instance, a published event can be consumed by more than one event sink.

Listing A.1: Clock System in CCM

```
eventtype tick {
  ...
}

interface Display {
  void update(...);
  ...
}

component Display {
  attribute int vertical;
  attribute int horizontal;
  attribute float minRefreshRatio;
  provides Display update;
  consumes tick synchronize;
}

home Displayhome manages Display {
  factory new(...);
}

component Clock {
  attribute int hours;
  attribute int minutes;
  attribute int seconds;
  uses Display informer;
  publishes tick ticker;
}

home Clockhome manages Clock {
```

```
  factory new(...);
}
```

Component integration [design with deposit-only repository].
CCM components are stored into a component repository, available in the application server. Component integration is done through method and event delegation. As we can see in figure A.4, event sources are matched to event sinks (`ticker` and `synchronize`, respectively), while facets are matched to receptacles (`update` and `informer`, respectively). At design time, developers specify how component instances should be wired using a Component Assembly Descriptor, which is an XML file such as the one in listing A.2, where some details are omitted.

Listing A.2: Clock System in CCM

```
<?xml version = ''1.0''?>
<!DOCTYPE component assembly CLOCKSYSTEM ''clocksystem.dtd''>
  <component assembly id = ''clocksystem''>
    <componentfiles>
      <componentfile id = ''Clock component''>
        <filearchive name = ''Clock.csd''>
      </componentfile>
      ...
    </componentfiles>
    <partitioning>
      <homereplacement id = ''ClockHome''>
        <componentfileref idref = ''Clock Home''/>
        <componentinstantiation id = ''clock''/>
        <registerwithnaming name = ''ClockHome''/>
      </homereplacement>
      ...
    </partitioning>
    <connections>
      <connectinterface>
        ...
      </connectinterface>
      <connectevent>
        <publishesport>
          <publishesidentifier>ticker</publishesidentifier>
          <componentinstantiationref idref=''clock''/>
        </publishesport>
        <consumesport>
          <consumesidentifier>synchronize</consumesidentifier>
          <componentinstantiationref idref = ''display''/>
        </consumesport>
      </connectevent>
    </connections>
  </component assembly>
```

During the deployment phase, the design time assemblies are not retrievable as

an entity, but component instances are integrated in the same configuration that was specified in design time. The instances run in the CCM container.

Model structure [flat].
CCM is a flat component model.

Contract support [basic syntax/type contract].
The CCM provides no direct support for design by contract, other than the contract that results from the IDL definition of the components. However, some extensions to the CCM, such as CIAO [Wang 04], built on top of the CCM, provide extra support for deploying components in real-time environments, where non-functional properties are crucial to the success of component integration. CIAO uses component composition metadata to allow QoS provisioning policies to be specified, decoupled from the component implementations.

Support for certification [not available].
The CCM provides no direct support for certification of components and component assemblies.

Support for compositional reasoning [not available].
The CCM provides no direct support for compositional reasoning.

A.4.6 Fractal

Origin [both].
Fractal is a component model independent from the programming language in which components are implemented. It is a hierarchical component model. Fractal components have reflective capabilities which are not fixed in the model, but can be extended and adapted to fit the developer's needs [Bruneton 04]. This allows developers to introduce non-functional aspects into component assemblies, for instance. The component model is developed by the Object-Web consortium[5], a non-for-profit consortium of organizations and individuals dedicated to the development of open source middleware.

Component representation [object].
A Fractal component is a runtime entity which, at the lowest level of control, does not provide any control capability to other components, thus behaving like an object. Objects are treated as components, which allows integrating components with legacy software in a convenient way. Components with the lowest level of control are called **base components**.

[5] `http://www.objectweb.org/`

At the second level of control, called **introspection**, Fractal components can provide a standard interface, similar to COM's **IUnknown**, to allow discovering all their services.

The third level of control is called **configuration**, and allows components to provide control interfaces that can be used to introspect and modify the component's **content**. These contents are other Fractal components, integrated through **bindings**.

Component syntax [programming languages with IDL mappings].
A Fractal component has a **component interface** which acts as its access point. The component implements a **language interface**, which is a type. Both interfaces are specified through an IDL. The rationale is to have mappings from the IDL to specific programming languages, and a compiler to generate component stubs and skeletons in those programming languages. For instance, in the Fractal specification, a pseudo-IDL is used, to denote that Fractal is implementation language-agnostic, but IDLs with mappings for Java, C, as well as the OMG IDL, are suggested for using Java, C, or any language for which OMG IDL has a mapping as the implementation language of components.

Figure A.5 represents the clock example, in Fractal. In this diagram, components are represented by double rectangles, where the controller part is shown in gray, and the content part is shown with a white background. The controller part establishes the links between the component's external interfaces, used to interact with other components, and internal interfaces used to interact with the component's sub-components.

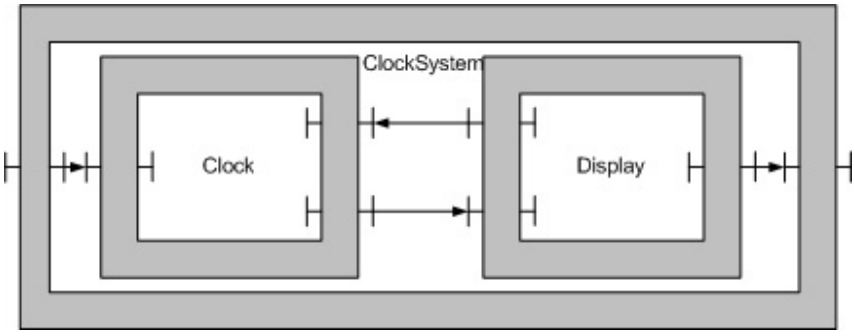


Figure A.5: Fractal components

Component integration [design without repository].
In the design phase, Fractal components are integrated through method calls, represented in figure A.5 as connectors. In the deployment phase, there is no assembler we can use to change the component integration. However, the Fractal Component model has dynamic reconfiguration capabilities, which are nevertheless configured at design time.

Contract support [basic syntax/type contracts].
The Fractal component model has a distinction between **functional** and **control interfaces**. The former are used for specifying the functionalities of components, while the later are dedicated to non-functional aspects. As such, the component model provides the basic information for contracts that address non-functional aspects. That said, Fractal provides no contract mechanism, as a default, other than the basic syntax/type contracts. However, support for design by contract can be added to Fractal by using ConFract, a contracting system that relies on an assertion language inspired in OCL, which allows defining contracts on Fractal components and interfaces [Chang 07]. ConFract supports both interface contracts, upon the required and provided interfaces, and composition contracts, which are built on the external and internal sides of components, to constrain the usage and internal assembly of components, respectively. Furthermore, ConFract enables non-functional contracts negotiation, including the propagation of such contracts in the hierarchy of components used in the component assemblies.

Support for certification [not available].
Fractal provides no specific support for the certification of software components.

Support for compositional reasoning [not available].
Although Fractal does not support compositional reasoning *per se*, this support can be added to Fractal, as Fractal is a modular component model which, by design, is extendable. Again, some forms of compositional reasoning (e.g. concerning resources capacity) are supported through the adoption of the ConFract contract system.

A.4.7 OSGi

Origin[industry].
The OSGi Alliance [6] was founded in 1999 and is promoted by a consortia of over 50 organizations (mostly companies). It aims to provide a service-oriented component-based environment for facilitating the interoperability of applications and services. It can be used to specify from fine to coarse-grained components. OSGi relies on the usage of Java for ensuring hardware portability.

Component representation [object].
OSGi has two main sorts of components: **bundles** and **services**. Each bundle contains a set of services.

Component syntax [programming languages with IDL mappings].
An OSGi service component has a set of interfaces and an implementation for those

[6] http://www.osgi.org/

interfaces. OSGi bundles have three kinds of ports:

- static ports, for allowing interaction with non-component-based software, namely through Java packages import and export;
- dynamic ports, for providing services to, or requiring services from, the environment; these services can be added to or removed from bundles at any moment;
- run-time environment ports, from which the bundles can learn about the availability of new services, therefore taking appropriate actions for the architecture to evolve, over time.

A bundle encapsulates service components, resources (such as images), and Java packages. Physically, the bundles are represented as *JAR* archives.

Component integration [design with deposit-only repository].
A main contrast between OSGi and most of the other component models is that while the latter typically support the assembly of known components, either in design, or in deployment time, and assuming the assembly to be static. OSGi assumes the system as an evolving set of components. So, when specifying a component, one is concerned with how components may be connected and disconnected at run-time, rather than in connecting components which are known *a priori*. The rationale is that OSGi bundles should be able to choose from the available components (other bundles) those that best match their necessities. In order for an OSGi bundle to be activated, all the packages it requires from the environment, including those provided by other bundles, must be available.

That said, the rules that dictate how components can be integrated are setup at design time, thus making OSGi integration a design with deposit-only repository.

The integration mechanism is flexible enough to allow for hot reconfiguration of components. For instance, when a backward compatible version of a bundle is made available, the component infrastructure provides the mechanisms so that the clients of the former bundle switch to the new component, while the whole system is running.

Model structure [flat].
The OSGi component model is flat.

Contract support [non-functional properties contracts].
OSGi bundles support basic syntax/type contract through the interfaces of the services provided and required by the bundles.

Through the bundles' meta-information, it is also possible to specify non-functional properties and use them to select, among other available bundles, the ones that conform to the requirements imposed by a bundle that is looking for other bundles to

support its own required interfaces.

Support for certification [not available].
OSGi provides no direct support for the certification of software components.

Support for compositional reasoning [not available].
OSGi provides no support for compositional reasoning.

A.4.8 Web services

Origin [both].
Web services are the composition units of a Web Service Architecture (WSA), and provide a standard interoperation mechanism for different software applications, running in different platforms and frameworks [W3C 04]. The WSA standards evolution is steered by a working group of the World Wide Web Consortium (W3C), whose mission is *"to lead the World Wide Web to its full potential by developing protocols and guidelines that ensure long-term growth for the Web"*. The consortium features organizations both from industry and academia.

Web services can range from fine to coarse-grained components. This results from the ability to compose web services, where a coarse-grained service can then delegate parts of its functionalities to other Web-services.

Component representation [object].
A Web service is a piece of binary code *"designed to support interoperable machine-to-machine interaction over a network."* [W3C 04].

Component syntax [programming languages with IDL mappings].
Each Web service has an interface described in a machine-processable format known as Web Service Description Language (WSDL). The service itself is implemented in a general purpose programming language, such as Java, or C♯, and made available in a server. The server makes the public interfaces of the Web services available in a registry of Universal Description, Discovery, and Integration (UDDI) identifiers.

Component integration [design with repository].
In Web services, there is no rigid architecture: services are used and composed through the usage of coordination languages and message passing. Web services and other client systems can interact with web services by exchanging Service Oriented Architecture Protocol (SOAP) messages. This interaction has to adhere to the Web service's WSDL interface. Typical message passing is achieved by using HTTP with an XML serialization, in conjunction with other web related standards [W3C 04].

In the design phase, a service can be composed with another service by specify-

ing the server where the latter is located, so that the former can send the latter SOAP messages (and *vice-versa*).

The deployment of services is also specified during design. At runtime each service runs in its own server, and the server is responsible for creating an adequate runtime environment.

Another form of composition is the one resulting from web services orchestration, and is performed on the client side, while creating client applications that use web services. These orchestrations are specified in Business Process Engineering Language (BEPL), to describe the workflows of those applications.

Model structure [hierarchical] Although web services do not have a rigid architecture, they can be hierarchically composed. A typical example is to have a top level composite service that can be decomposed into several task level services. In turn, each of these services can be further refined into several different sub-tasks, and so on.

Contract support [non-functional properties contracts].
The WSDL interface description provides the basic support for contract specification in Web services. The WSDL specification of a Web service defines how the Web service will handle incoming SOAP messages. This includes not only the message formats and typing information, but also the transport protocols and serialization formats required for the communication between services.

The support for contracts goes well beyond basic static type contracts: Web services may provide a contract describing the terms and conditions of the services they provide. This contract is known as Service Level Agreement (SLA) and provides Quality of Service (QoS) information to Web services users, such as **execution price** and **duration**.

Support for certification [not available].
The Web services standard provides no direct support for the certification of Web services.

Support for compositional reasoning [not available].
Web services provide no support for compositional reasoning.

A.4.9 Acme

Origin [academic].
Software architectural descriptions provide an abstract representation of the components of software systems and their interactions. These descriptions may range from ad-hoc notations to formal ADLs. The latter usually support some level of analysis with respect to the consistency and completeness of the component-based systems they

model.

There are several examples of ADLs, such as Aesop [Garlan 94], Adage [Coglianese 93], C2 [Medvidovic 96], Darwin [Magee 95], Rapide [Luckham 95], SADL [Moriconi 95], UniCon [Shaw 95b], MetaH [Binns 93], or Wright [Allen 97]. Although with a considerable overlap on the core, each ADL focuses on different aspects of software architecture. This diversity provides different approaches to solve specific families of problems, but the difficulty in interchanging information between different ADLs was a major drawback. Developing a single ADL providing all the features of the various ADLs would be a very complex endeavor. Instead, an ADL called Acme [Garlan 00b] was proposed as a generic language which can be used as a common representation of architectural concepts in the interchange of information between specifications with different ADLs. The rationale is that, rather than developing mappings for all combinations of ADLs, the ADLs community only needs to provide mappings from other ADLs to Acme and vice-versa [Barbacci 98]. Acme components can range from fine to coarse-grained components.

Component representation [architectural units].
In Acme components are architectural units. See the component syntax for further details on Acme components.

Component syntax [ADL].
Figure A.6 illustrates the graphical presentation of an Acme description, where some of Acme's design elements are used. Both components (`Clock` and `Display`), have two ports and a set of properties. The components are integrated using two connectors, `tick` and `info`. The architecture description is further detailed in Acme's textual description (listing A.3) of the same architecture, where we added some properties to the specification of the connectors. Note that several details are ommitted in both the graphical and textual descriptions, for the sake of brevity.

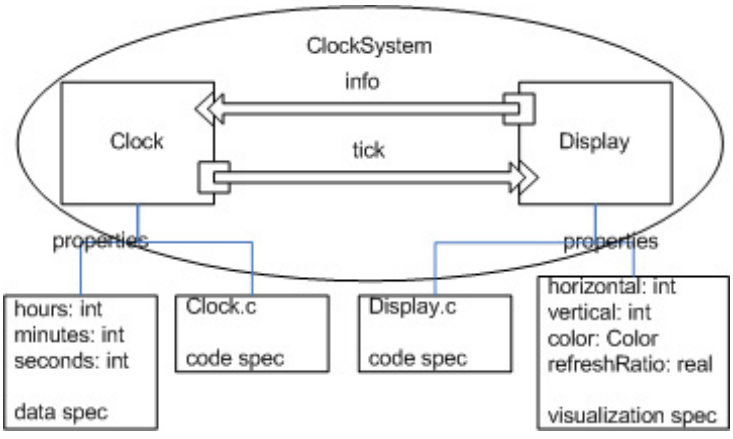


Figure A.6: A simple clock system in Acme

Listing A.3: Clock System in Acme

```
System ClockSystem = {
  Component Clock = {
    Port ticker;
    Port informer;
    Properties { hours : integer;
                 minutes : integer;
                 seconds : integer;
                 sourceCode : externalFile = ``Clock.c''}
  }
  Component Display = {
    Port update;
    Port synchronize;
    Properties { vertical : integer = 300;
                 horizontal : integer = 400;
                 color: Color;
                 RefreshRatio: real;
                 sourceCode : externalFile = ``Display.c''}
  }
  Connector tick = {
    Role source;
    Role sink;
    Properties { synchronous : boolean = false ;
                 maxRoles : integer = 2 ;
                 protocol : Wright = ``...'' }
  }
  Connector info = {
    Role caller;
    Role callee;
    Properties { synchronous : boolean = true;
                 maxRoles : integer = 2 ;
                 protocol : Wright = ``...'' }
  }
  Attachments : {
    Clock.ticker to tick.source ;
    Display.update to tick.sink ;
    Display.synchronize to time.caller ;
    Clock.informer to time.callee ; }
}
```

An Acme **component** has **ports**(e.g. `ticker`, in the `Clock` component), which act as the component interfaces, **properties** (e.g. `hours`, in the `Clock` component), a **representation** with several bindings (defined as rep-maps) and a set of **design rules**. Neither rep-maps nor design rules are represented in this example, for simplicity.

Acme ports identify points of interaction between a component and its environment. They can be as simple as operation signatures, or as complex as collections of procedure calls with constraints on the order in which they should be called. Acme

ports can only be used with Acme components and they have one provided and one required interface.

Acme **connectors** represent interactions among components. They are viewed as first class elements in this ADL. Acme connectors may be much more complex than a simple interfaces' match. They can be, for example, a protocol, or a SQL link between two components (e.g. a client and a database). The `tick` connector is asynchronous. The `info` connector is synchronous. Both connectors have two roles, and a protocol specification property defined here using a different ADL (Wright [Allen 97]). When reusing components built by different teams it is normal that their interfaces do not match exactly. The connector may provide the necessary glue between the components and this must be made explicit in the design. Roles are related to connectors the same way as ports are related to components.

An Acme **system** represents a graph of interacting components. Acme's representations provide the mechanism to add detail to components and connectors. Rep-maps are used to show how higher and lower-level representations relate to each other. Acme explicitly uses the concepts of representation and system for defining subsystems.

Properties represent semantic information about a system and its architectural elements. Among the properties of components, we can highlight the property concerning the source code of each of the component. In this example the **source code** is defined in the C programming language [Kernighan 88], but many other programming languages could be used, instead, as Acme is independent from the specific programming language used in the implementation of the components.

Ports can be typed with a provided interface that allows the component user to access its properties.

Component integration [design without repository].
Acme does not have the notion of a repository for its architectural elements. All these elements have to be specified from scratch, when designing an architecture. The architecture may be specified using a visual tool such as AcmeStudio[7]. The architecture specification can then be built into a system, as long as the source code for components and connectors is available. Acme architectures can be specified in ArchJava [Aldrich 08] (which is part of the AcmeStudio) and then compiled into Java, and run on the Java Virtual Machine.

Model structure [hierarchical].
The component model is hierarchical. In other words, fine-grained components can be composed into coarser-grained components, and so on. For instance, a component-based system may be used as a component in a larger system.

[7] `http://www.cs.cmu.edu/acme/`

Contract support [non-functional properties contracts].
Acme supports a mechanism for specifying and checking **constraints**. This allows the specification of claims on how the architecture and its components are supposed to behave. Acme supports two sorts of constraints: **invariants** and **heuristics**. While invariants are conditions that must hold at all times, heuristics are constraints that should hold, although breaking them is possible. An **architectural style**, or **type**, defines a vocabulary of design elements and the rules for composing them. It is used in the description of **families of architectures**.

Constraints may be automatically verified. For example, AcmeStudio checks for violations of design constraints [Garlan 03]. The added flexibility brought by the possibility of embedding specifications of contracts in other ADLs (e.g. in Wright, in the clock system example) makes it possible to support any of the levels of Beugnard's contracts.

Support for certification [not available].
Acme provides no special support for third party certification of components and systems, other than the constraints checking mechanism. For instance, consider the `requestRate` property defined for the *client* component. Unless the property can be ensured to hold by construction (e.g. through the usage of a code generator that enforces that property), Acme provides no direct support to ensure that the implementation of the component will always respect this constraint. If a the component's implementation has an unknown bug that violates this constraint, this will not be detected through the Acme specification.

Support for compositional reasoning [available].
As long as the implementations of the architecture artifacts fulfill their specified properties, model checking techniques can be used to certify derived properties of the whole system. While compositional reasoning is possible for some properties, through the definition and observance of model constraints, its applicability to other properties may depend on external factors. For instance, in the clock system, the protocol is been specified in Wright. As such, the support for compositional reasoning with respect to that protocol is the support provided by the Wright ADL, rather than the direct support of Acme. Model checking techniques have some limitations, particularly in what concerns their scalability. Furthermore, there are unsolved issues concerning dynamism [Garlan 03], as modern architectures are no longer static. Consider, for example, architectures with support for plug-ins that can be connected at run-time. ADLs such as Acme are not well suited to represent such architectures.

A.4.10 UML 2.0

Origin [both].
UML is a standard proposed by the OMG for modeling software systems. Although it is best known by its usage in modeling use cases, for capturing requirements, and class diagrams, to support object-oriented analysis and design, UML has also support for component-based development, among several other features. In UML 2.0 [OMG 05b,OMG 06b], OMG has significantly improved the support for modeling software components by incorporating in UML 2.0 several abstractions borrowed from other ADLs.

The OMG team responsible for steering the evolution of UML has a balanced composition, including experts from both the industry and the academia, granting UML users the availability of a wide range of professional tools, along with scientific *fora* (e.g. the MoDELS conference series[8], formerly known as the UML conference series) where UML's foundations and evolution are discussed.

UML components can have any granularity, ranging from fine grained components to full application-like components.

Component representation [architectural units].
UML's representation of software components is comparable to that of other ADLs, such as Acme [Goulão 03]. Components are first class architectural elements. In contrast with other ADLs, connectors are not first class architectural elements.

Component syntax [ADL].
UML 2.0 features new component diagrams, when compared to previous versions of the UML. Figure A.7 presents an example of a composite structure with the components Clock and Display, each of which with a provided and required interfaces. In this example, components are presented as black-boxes, with a set of ports typed by interfaces. UML 2.0 is flexible in the definition of the relationships between components and interfaces. Interfaces can be directly provided, or required, by the component, without using a port.

Component integration [design without repository].
UML 2.0 components can be composed into coarser-grained components. In this sense, the UML 2.0 component model is hierarchical. The details of this integration can be made explicit in UML 2.0 component diagrams, by presenting a white-box view of components. UML 2.0 provides a delegation is a connector that is used to specify that the behavior available in a component instance is not realized by that component, but rather by another component, or class, which has compatible capabilities.

The other form of integrating components is through an assembly connector. It

[8] `http://www.models.org/`

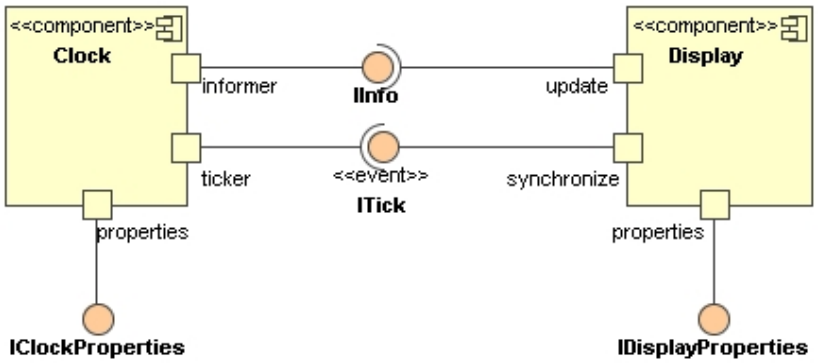


Figure A.7: UML 2.0 components

is represented by the lollipop notation, adapted from COM+ and CCM, and is used to decorate whether an interface is being provided (the "lollipop") or required (the "lollipop" plug) by a component. Note that the `ITick` interface is used so that an event generated by the `Clock` component can be captured by the `Display` component. In UML 2.0, this can be done by stereotyping an interface feature with `<<event>>`.

Unlike Acme, the assembly connector is essentially a wire connecting a provided with a required interface, rather than a first class design element representing a connector where, for instance, a communication protocol could be defined.

UML 2.0 offers no support with respect to a repository of components. They are created and wired at design time, and that wiring can be used in code generation for a given programming language, but components cannot be assembled in deployment time.

Model structure [hierarchical]

UML allows the hierarchical decomposition of components into simpler components, as well as to the implementation classes. So, it is common to see fine grained components implemented with a set of classes, and an assembly of those components into medium grained components, which in turn are used as the building blocks for larger components.

Contract support [behavior contract].

UML 2.0 includes the object constraint language (OCL), that allows specifying constraints upon a UML model. OCL offers support for defining invariants on the model and model elements, as well as pre and post conditions on operations. These constraints can be used, for instance, to enforce a given architectural style (see, for instance [Goulão 03]), as well as to create a design by contract expressiveness comparable to that of Eiffel. Furthermore, it is possible also to use other UML 2.0 diagrams to complement these restrictions, allowing for a more expressive contract definition. For instance, sequence diagrams can be used to define behavioral aspects of ports.

Support for certification [not available].
UML 2.0 offers no special support for the certification of components.

Support for compositional reasoning [not available].
UML 2.0 offers no special support for compositional reasoning.

A.4.11 Kobra

Origin [both].
The KobrA component model [Atkinson 01] is a UML-based component model that integrates CBD with software product lines. Its development has been lead by Fraunhofer IESE (Germany). The KobrA approach encompasses not only a component model, but also the process for developing and exploring software product lines. The emphasis on product line engineering justifies the special care taken by its proponents to modeling commonalities, scope, and variability within the product line.

Component representation [architectural units].
The KobrA component metamodel is an extension to the UML 1.4 metamodel. As such, KobrA modeling elements are stereotyped UML model elements. KobrA components, known as **Komponents** are stereotyped UML subsystems. They differ from a UML subsystem in that components can have a behavior, unlike UML subsystems. So, while a subsystem has no behavior *per se* (it is a repackage of the behavior of its internal model elements), a component can add new services, which typically use the services made available by the component's internal elements. We can also describe this contrast as the one between a module used for organizing and packaging its internal constituents, or a component which adds its own behavior to that of its constituents. A complete system may also be modeled as a component.

Component syntax [ADL].
A component specification may contain up to six specification artifacts:

- **Structural model**. The structure specification of a component specifies its visible interfaces and structure, and its internal structure, both in terms of classifiers and of object instantiations. A structural specification includes at least a class diagram, but may also include object diagrams, to specify the internal instantiation of objects belonging to the component.

- **Behavioral model**. The behavioral model specifies how the component is expected to behave in response to external stimuli. The specification itself is performed using UML statechart diagrams, or statechart tables.

- **Functional model**. The functional model specifies the operations made available through the component's interfaces. This specification includes a general description of the operation, of what the operation receives from and sends back to other components, the invariants that are preserved, what is changed by executing the operation, what are the assumptions made before executing the operation, and which are the expected results after its execution. In practice, the functional specification provides a contract with the usual pre- and post-conditions, as well as invariants (all of which can be described in natural language, or, in a more formal alternative, OCL).

- **Non-functional requirements specification**. This specification is optional and provides statements constraining the options of component developers, with respect to the non-functional properties of a component. For instance, one such constraint could be that a component must be implemented in Java. The KobrA approach does not prescribe exactly how these requirements should be specificed.

- **Quality documentation**. The quality documentation is optional in KobrA and should cover quality modeling based on structural properties of KobrA components. The rationale is that these structural properties are available early in the development cycle and its measurement can be useful for detecting and correcting potential problems early. The quality model used in the KobrA approach relates structural properties (internal attributes) such as coupling, complexity, and size, with externally visible quality attributes, such as ISO 9126 [ISO9126 01] quality attributes like reliability and maintainability.

- **Decision model**. The decision model constrains the selection of variations for the valid configuration of products.

The first three specifications are mandatory, and capture the functional requirements of the software components. The last three specifications are optional and relate to non-functional aspects of the components, which influence component's quality and help capturing product-line engineering concerns [Atkinson 01].

The KobrA approach encompasses not only the specification of components, as discussed above, but also its realization. The latter includes the development of a set of artifacts: structural model, activity model, interaction model, data dictionary, quality documentation, and decision model. These artifacts can be contrasted with those described for the specification of the component. While in the specification the concern is on *what* the component is supposed to do, in the realization the concern shifts to *how* the component is supposed to do it. The implementation of components is not tied to a particular programming language, although it has to accommodate the constraints described in the specification and refined in the realization.

Figure A.8 presents a simplified view of a structural diagram focused on the `Clock` component. The component depicted in the diagram is stereotyped with `<<subject>>` while other cooperating components are stereotyped with `<<Komponent>>`.

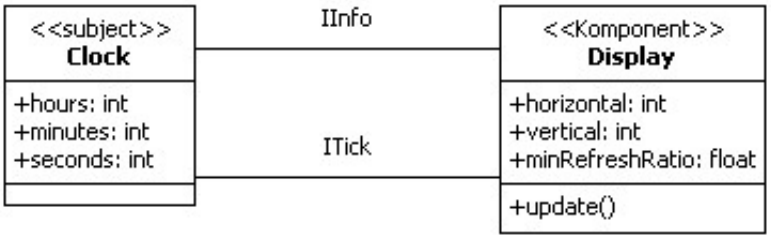


Figure A.8: KobrA structural diagram

Component integration [design with repository].
KobrA components can be constructed using a UML tool and deposited into the file system, which acts as the component's repository. Composition occurs only in design time. In deployment time, component's specifications can be refined in the component's implementations, but no further composition is possible.

There are three kinds of relationships between components:

- **Composition** corresponds to a link between a composite component and its parts, established at runtime. It includes the creation of the composite component's parts, and therefore creates a clientship relationship between the composite component and its constituents.

- **Clientship** corresponds to a link between two component instances, which is also established at runtime. It is a unidirectional relationship, in the sense that the client component needs to know the identity of the server to access its provided services, but the server is unaware of the client.

- **Ownership** KobrA componetns can be regarded as modules, or packages, which may contain other model elements (including other components). This containment relationship is called Ownership.

Component model structure [hierarchy] KobrA components may range from small to coarse-grained components, and this is supported by the hierarchical nature of KobrA's component model (see the Ownership relationship, in the previous section). In this component model, complete systems are represented as a component.
Contract support [behavioral contract].
KobrA provides support for structural and behavioral contracts. The former is achieved through the interfaces of the operations offered by the component's interfaces, along with the specification of those operations, while the latter is made

available through the specification of the operation's behavioral model. The collection of behavior models for all the operations provided in the component interfaces constitute the component's functional model.

Support for certification [not available].
The KobrA component model provides no special support for the certification of components by a third party. However, the KobrA development process does prescribe the development of functional test cases that are stored along with the component. These test cases should be created with respect to the component's specification.

Support for compositional reasoning [not available].
The KobrA component model provides no special support for compositional reasoning. Again, the KobrA development process mitigates this shortcoming through the usage of a technique called Extreme Harvesting [Atkinson 05], which is basically a process for looking up software components in a large component repository (e.g., the world wide web!). The process includes the definition of a syntactic signature of the component, a definition of its semantics, in terms of test cases it must conform to, and the search of those components in source code format using a normal search engine (e.g. google). The components that are an exact match for the specified signature and are able to pass the test battery created for verifying their semantics are considered candidate components for composition. Note that this method for detection ignores whatever is not tested in the test suite (e.g. non-functional requirements).

A.4.12 Koala

Origin [industry].
The Koala component model [Ommering 00, Ommering 04] was proposed in the Philips Research Laboratories. It is a component model developed for supporting the development of embedded software for consumer electronic devices, and currently used in the development of Philips's software product lines (SPL). In Koala's development context, it was particularly important to support the development of product families (e.g. TV sets) and product populations (e.g home entertainment electronics devices). A SPL as *"a proactive and systematic approach for the development of software, to create a variety of products"*. A product family is *"a set of products with many commonalities and a few differences"*. A product population is *"a set of products with many commonalities, but also many"* (the definitions of SPL, product family and product population are quoted from [Ommering 04]).

The granularity of Koala components may range from fine-grained to coarse-grained components. As noted by van Ommering [Ommering 04], a typical Koala component implements about 10 interfaces, and a component assembly (configuration, in Koala's terminology) usually has tens of components.

Component representation [architectural units].
Koala components are units of design, development, and reuse. They are built to be strictly separated from configuration development. Component developers make no assumptions on the configurations upon which components are used, and component users have no way of changing the components to fit their needs. We can regard components as architectural units, due to the ADL-like nature of the languages used to specify them, as we will discuss in the component syntax section.

Component syntax [ADL].
Koala components have a specification and an implementation, each expressed in its own language. The specification is expressed in a combination of three ADLs: one for specifying the component's interfaces, another for specifying the component, and yet another for specifying the local data of components.

Interfaces are defined in the **I**nterface **D**escription **L**anguage (**IDL**), which has a syntax similar to C. For instance, the `Clock` component would implement the interfaces `IInit` and `IClock`. The former is dedicated to the component initialization methods and common to many other components, while the latter is dedicated to the operations which are specific to the `Clock` component.

Listing A.4: Clock System in Koala

```
interface IInit {
  * component initialization methods *
}

interface IClock {
  void setHours(int hours);
  void setMinutes(int minutes);
  void setSeconds(int seconds);
  int getHours(void);
  int getMinutes(void);
  int getSeconds(void);
}
```

The boundaries of components are defined in a textual component description language (CDL). Assuming the clock component is to be wired to a display component which will provide the IDisplay interface, we can define the component as CClock in CDL as follows:

Listing A.5: Clock System in Koala

```
component CClock {
  provides IClock   pclock;
           IInit    pini;
  requires IDisplay pdisplay;
}
```

Koala components are compiled into the a programming language. Although the component model is independent from the chosen programming language, the language chosen for the initial implementation of the tools that support the development in Koala is C. So, Koala components are used to define the corresponding C header files. The implementation of those header files is then performed in C by the component developers.

Component integration [design with repository].
In Koala, components can be integrated by specifying a component configuration. Koala's component configurations have to follow some constraints: each and every required interface has to be bound to exactly one provided interface, while a provided interface has to be bound to zero or more required interfaces. The Koala component model is hierarchical, so, it is possible to define compound components. The components are stored into a component repository. Component configurations are created at design time, as composite components which are also stored in the component repository. At deployment time, no further integration is available. The components are compiled into a programming language and run in that programming language's run-time environment.

Model structure [hierarchical].
Components may be integrated into compound components, making this component model hierarchical.

Contract support [basic syntax/type contract].
Koala supports the basic syntax/type contract, through the usage of the component interfaces.

Support for certification [not available].
Koala has no specific support for component certification.

Support for compositional reasoning [not available].
Koala has no specific support for compositional reasoning.

A.4.13 SOFA 2.0

Origin [academy].
SOFA is an academic component model proposed in [Plásil 98], that has evolved since then The original goal of the component model was to allow updating software at runtime, in a way that could be integrated with the electronic commerce of software components. SOFA's authors decided to create a new version of it (2.0) that allows dealing with the dynamic reconfiguration of component assemblies, creating a struc-

ture of the control part of the component, and supporting multiple communication styles between components [Bures 06, Bures 07].

The hierarchical nature of SOFA components provides support for developing components ranging from small to coarse-grained.

Component representation.
A SOFA component is an architectural unit with its corresponding specification and implementation.

Component syntax [ADL].
SOFA components are described by a **frame** and an **architecture**. The frame corresponds to a black-box view of the component, with its provided and required interfaces, as well as the component's properties. The architecture is a corresponds to a gray-box view of the component that implements the component's frame by using sub-components and their inter-connections, in a first level of nesting. A frame can be implemented by more than one architecture, which introduces a possible variation point in the composite component: an implementation can be replaced by another one, while maintaining the same frame, and the choice of which implementation is used can be performed in assembly time, rather than design time alone.

SOFA components are interconnected through connectors, which are first class entities. Component behavior can be formally specified through behavior protocols. Figure A.9 presents an example of a SOFA architecture where the `Clock` and `Display` components are integrated in a `ClockSystem` component. The black rectangles represent component's provided interfaces, while the white rectangles represent the required interfaces. The lines connecting components are connectors. In SOFA 2.0, these are first-class entities, and can be used with several purposes:

- as delegations between a component and a sub-component, when a sub-component provides an interface that is externalized by the component, the component delegates the implementation of the interface to the sub-component;
- as delegations between a sub-component and the component, when the sub-component requires an interface, and that requirement is delegated to the component which will expose that interface as his own required interface;
- as a bound between the required interface of a component and the provided interface of another.

SOFA connectors allow incorporating more than one interface in the same communication link (e.g. connecting one server to several clients), and have support for directly expressing different architectural styles, such as pipe and filter, bus communication and shared memory (in contrast to UML, for instance where the styles would be

implemented through components, thus mixing "utility" components with "business" components) [Bures 07].

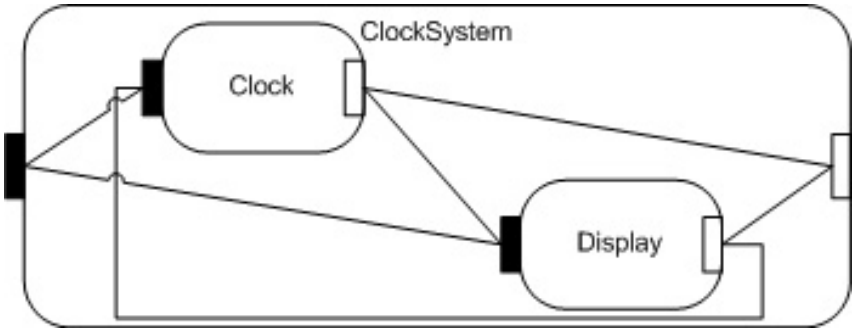


Figure A.9: SOFA 2.0 components

Components are specified using an ADL called **C**omponent **D**efinition **L**anguage (**SOFA CDL**). SOFA CDL definitions are then compiled to a programming language (Java) that provides the runtime environment for the components.

Component integration [Architectural unit].
SOFA components can be either primitive or composite. Composite components are built of other components, but cannot add new business functionality to the one provided by their sub-components. So, at development time, component developers can only develop the source code of primitive components, and then integrate the new primitive components with other components from the repository. SOFA is an hierarchical component model. The following code illustrates the textual representation of a composite component. In this example, we use two different kinds of predefined connectors, `CSProcCall` and `EventDelivery`, for synchronous and asynchronous communication between the components, respectively.

Listing A.6: Clock System in SOFA.

```
system CUNI ClockSystem version ``1.0'' {
  inst Clock aClock;
  inst Display aDisplay;
  bind aDisplay.update to aClock.informer using CSProcCall;
  bind aClock.ticker to aDisplay.synchronize using EventDelivery;
  ...
```

Component integration can be performed at the design time, by linking components with connectors and storing the components into a repository. Both the primitive and composite components can be stored in the component repository.

Furthermore, SOFA supports the dynamic evolution of components, mainly by creating three evolution patterns: a **factory pattern**, for creating new components in the component assembly, a **removal pattern**, for deleting the dynamically created components, and a **service access pattern** for allowing the creation of utility interfaces, which are exposed as services that can be registered in a service registry. This allows utility

interfaces to be dynamically bound and unbound at runtime, through requests sent to the service registry. This registry can be made available both to other components applications and even non-component-based applications [Bures 07].

At runtime, the application is deployed to SOFA's runtime environment, which is called **SOFAnode**. The SOFAnode includes a component repository and deployment docks, which are basically containers that provide the infrastructure for starting, stopping and updating components. SOFAnode is implemented Java, so, *"under the hood"*, SOFA is using the Java Runtime Environment.

One of the distinguishing features of SOFA is the ability to define **controllers** as micro-components which are organized as component aspects. These controllers implement the SOFA control logic. When an architecture is deployed, the code of the components is weaved with that of the controllers, so that the application can be run using a selected control logic. The basic query, binding, and life-cycle logic is implemented this through a set of such micro-components.

Model structure [hierarchical].
SOFA is a hierarchical component model.

Contract support [non-functional properties contracts].
SOFA supports both the syntactical contract provided through the interfaces, and the definition of behavior protocols, which, in practice, can be regarded as behavior contracts. Indirectly, through the definition of controllers, it may also support other, more sophisticated levels of contracts. Having connectors as first-class architectural entities provides flexibility for defining, for instance, performance measurement probes in connectors, which can be used for ensuring non-functional properties. The usage of micro-components can also help implementing such contracts.

Support for certification [not available].
SOFA provides no direct support for the certification of components.

Support for compositional reasoning [available].
The behavior protocols used in SOFA provide the basis for performing compositional reasoning. The SOFA team is currently addressing this problem by developing a **Behavior Protocol Checker**, which tries to handle the typical state explosion problem of these checkers through the usage of parse tree automata, which are subject to several optimizations to trim their size, namely multinodes, forward cutting, and explicit subtree automata [Mach 05].

A.4.14 PECOS

Origin [both].
The PECOS component model [Winter 02] was developed in the context of a European project including both industry and academic partners. The model was developed with a particular focus on embedded component software, for small field devices, which have typically hard real-time constraints. These devices typically use sensors to collect data from their environment and then process it to react accordingly, by controlling actuators, such as motors.

According to [Winter 02], the typical devices that run PECOS software have tight resource constraints, such as their available memory, as well real-time constrains. As such, a typical component in PECOS ranges from small to medium grained.

Component representation [architectural units].
PECOS components are represented as architectural units. They are units of design, with a specification and an implementation.

Component syntax [ADL].
PECOS supports three kinds of software components: **active components**, **passive components**, and **event components**. Active components have their own thread of control and are used to model long-going activities. For instance, a complete system is modeled as a composite active component. In contrast, passive components do not have their own thread of control and encapsulate behavior that executes synchronously and completes in a short cycle time. These components are scheduled by the nearest active parent that contains them. Event components are similar to active components, but their internal control subnet does not cycle. Instead, their functionality is triggered by an event. Event components can be used, for instance, for modeling hardware devices that periodically generate events. Common applications for these components include timers and components that periodically send status information about a device.

PECOS components have interfaces with a number of ports. A port is a shared variable used for allowing communication between different components. Connected ports represent the same variable. Connectors connect ports of compatible type, direction and range.

PECOS components have a specification, using an ADL called CoCo, and can be implemented in a general purpose programming language. CoCo allows specifying the structural aspects of the components, but not their behavior. The latter is left for the implementation.

Figure A.10 presents the `ClockSystem`, in PECOS. In this example, the clock system itself is represented as a component, to illustrate the hierarchal nature of this component model.

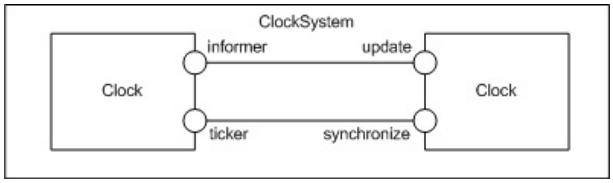


Figure A.10: The clock system, in PECOS

Component integration [design without repository].
The CoCo ADL allows specifying the composition of components into sub-components, in the design phase. As noted in the previous section, PECOS components are connected through ports, which have to belong to the same parent component (in other words, connectors may not cross component boundaries). The system, as a whole, is represented as a composite active component and normally represents a device which has an execution model that is, essentially, a loop.

Component model structure [hierarchy].
PECOS components can be hierarchically composed, so components can be classified as **composite**, or **leaf** components. Any of the component kinds discussed in the component syntax section (active, passive, and event) can be composite.

Contract support. [Synchronization contracts].
We can regard the support of PECOS components for the synchronization and timing of the interactions between components as a synchronization contract that allows, for instance to ensure deadlock-free interaction between components, and partial ordering for the execution of components within a composite component. PECOS components also support the basic kind of contracts. However, they do not support neither behavior contracts (as mentioned earlier, the implementation is separate from the CoCo specification), nor contracts for extra-functional properties.

Support for certification [not available].
PECOS provides no special support for certification.

Support for compositional reasoning [available].
The composition of PECOS components has an underlying execution model that deals with synchronization and timing issues. Composite components have as many threads of control as internal active components. The execution semantics of components is formalized as a Petri net interpretation [Nierstrasz 02].

Appendix B

Bridging the gap between Acme and UML for CBD

Contents

Background: *Acme is an Architecture Description Language (ADL) that contains the most common ADL constructs, and provides formality in the description of software architectures. The lack of stronger tool support hampers its adoption by a larger community, as it happens with several other ADLs. UML, on the other hand, has become a de facto standard notation for design modeling, both in industry and in academia.*

Objectives: *Our goal is to map Acme modeling abstractions into UML 2.0, to assess the expressiveness of the latter in what concerns Architectural description.*

Techniques: *We use a UML lightweight extension mechanism, with OCL well-formedness rules, to build a UML profile for Acme.*

Results: *The feasibility of this mapping is demonstrated through several examples.*

Limitations: *This mapping focuses on the structural aspects of Acme, but it does not cover the dynamic ones.*

Conclusions: *This mapping bridges the gap between architectural specification with Acme and UML, namely allowing the transition from architecture to implementation, using UML design models as a middle tier abstraction.*

B.1 Introduction

Software architectural descriptions provide an abstract representation of the components of software systems and their interactions. There are three main streams of architectural description techniques: ad-hoc, OO techniques and ADLs.

Ad-hoc notations lack formality, preventing architectural descriptions from being analyzed for consistency or completeness and for being traced back and forward to actual implementations [Garlan 00a].

To overcome those drawbacks, one can use ADLs, such as Aesop [Garlan 94], Adage [Coglianese 93], C2 [Medvidovic 96], Darwin [Magee 95], Rapide [Luckham 95], SADL [Moriconi 95], UniCon [Shaw 95b], MetaH [Binns 93], or Wright [Allen 97]. Although with a considerable overlap on the core, each ADL focuses on different aspects of architectural specification, such as modeling the dynamic behavior of the architecture, or modeling different architectural styles. This diversity provides different approaches to solve specific families of problems. However, the interchange of information between different ADLs becomes a major drawback. Developing a single ADL providing all the features of the various ADLs would be a very complex endeavor. Instead, an ADL called Acme [Garlan 00b] emerged as a generic language which can be used as a common representation of architectural concepts in the interchange of information between specifications with different ADLs [Barbacci 98].

Although ADLs allow for architecture in-depth analysis, their formality is not easily reconciled with day-to-day development concerns. OO approaches to modeling, on the other hand, are more widely accepted in industry. In particular, the UML has become both a *de jure* and *de facto* standard. Using it to describe software architectures could bring economy of scale benefits, better tool support and interoperability, as well as lower training costs.

OO methods have some advantages in the representation of component-based systems, when compared to ADLs. There is a widespread notation (UML), an easier mapping to implementation, better tools support and well-defined development methods. But they also have some shortcomings. For instance, they are less expressive than ADLs when representing connections between components.

Several attempts to map ADLs to UML have been made in the past, as we will see in section B.4. One motivation for such attempts is to bring architectural modeling to a larger community, through the use of mainstream modeling notations. Another is to provide automatic refinement mechanisms for architectures. UML can be used as a bridge from architectural to design elements [Egyed 01]. In this appendix we present a more straightforward mapping from Acme to UML, when compared to previous attempts, due to the usage of the UML 2.0 metamodel.

We represent the concepts covered by Acme using the UML 2.0 metamodel [OMG 06b, OMG 05b, OMG 03b]. This increases our modeling power due to the fea-

tures the UML 2.0 version, mainly in what concerns the representation of components, ports, interfaces (provided or required), and the hierarchical decomposition of components.

This appendix is organized as follows. Section B.2 contains a formal specification of the mapping between Acme and UML. Section B.3 includes a discussion of the virtues and limitations of that mapping. Related work is discussed in section B.4. Section B.5 summarizes the conclusions and identifies further work.

B.2 Mapping Acme into UML

For the sake of brevity, we omit the OCL definition of predicates such as `IsAcmeComponent()`, `IsAcmeConnector()`, `IsAcmePort()`, `IsAcmeRole()`, `IsAcmeProperty()` and others with self explanatory names that will be used in our mapping presentation. `HasNoOtherInterfaces()` is a predicate that denotes that no other interfaces except for the ones defined in ports will be available for a particular component.

B.2.1 Components

An Acme component has ports, which act as the component interfaces, properties, a representation with several bindings (defined as rep-maps) and a set of design rules. The closest concept to an Acme component in UML is the one of component. To avoid mixing Acme's components with other concepts that we will also represent with UML components, we created a stereotype for Acme components named `<<AcmeComponent>>`, using UML's `Component` as the base class. `Invariant 1` assures these components only have interfaces through Acme ports or properties (listing B.1.

Listing B.1: Invariant 1.

```
context Component inv: -- Invariant 1
  self.IsAcmeComponent() implies
    self.ownedPort->forAll(ap|
      ap.IsAcmePort() or ap.IsAcmeProperty())
    and self.HasNoOtherInterfaces()
```

B.2.2 Ports

Acme's ports identify points of interaction between a component and its environment. They can be as simple as operation signatures, or as complex as collections of procedure calls with constraints on the order in which they should be called. UML ports are features of classifiers that specify distinct points of interaction between the classifier (in this case, the component) and its environment (in this case, the rest of the system). UML ports have required and provided interfaces, which can be associated to pre and

post conditions. We use a combination of UML port and corresponding required and provided interfaces to express Acme's port concept. Acme ports can only be used with Acme components and they have one provided and one required interface. Listing B.2 specifies the well-formedness rule for ports.

Listing B.2: Invariant 2.

```
context Port inv: -- Invariant 2
  self.IsAcmePort() implies
     self.owner.IsAcmeComponent() and
    (self.required->size()=1) and
    (self.provided->size()=1)
```

B.2.3 Connectors

Acme connectors represent interactions among components. They are viewed as first class elements in the architecture community. Representing them using UML's assembly connector would be visually appealing, but we would loose expressiveness because Acme connectors may be much more complex than a simple interfaces' match. They can be, for example, a protocol, or a SQL link between two components (a client and a database). Moreover, when reusing components built by different teams it is normal that their interfaces do not match exactly. The connector may provide the required glue between the components and this must be made explicit in the design. In order to represent the concept of connector, which has no direct semantic equivalent in UML, we use a stereotyped component named `<<AcmeConnector>>` and ensure that it has no other interfaces than the ones defined through its roles and properties. Listing B.3 presents the well-formedness rule for connectors.

Listing B.3: Invariant 3.

```
context Component inv: -- Invariant 3
  self.IsAcmeConnector() implies
    self.ownedPort->forAll(ap|
      ap.IsAcmeRole() or
      ap.IsAcmeProperty()) and
    self.HasNoOtherInterfaces()
```

Although representing a connector with a stereotyped component clutters the outcoming design[1], it offers the ability to represent the connector as a first class design element, with flexibility in the definition of any protocols it may implement. Consider the example in Figure B.1, where the components client and server have interfaces that do not match, but the `rpc` connector implements a protocol to make both components interact. We have included provided and required interfaces in both ends of the connector, to illustrate that it provides bi-directional communication abilities.

[1]This cluttering problem can be circumvented by creating a different visual representation for these connectors, as supported by UML extension mechanisms.

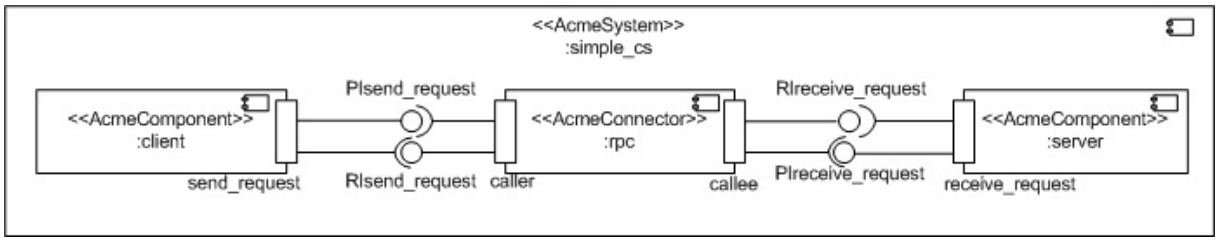


Figure B.1: Using the Acme connector

B.2.4 Roles

In Acme, roles are related to connectors the same way as ports are related to components. Thus, it makes sense to represent Acme roles as constrained UML ports, through the use of the `<<AcmeRole>>` stereotype. The corresponding well-formedness rule is presented in listing B.4.

Listing B.4: Invariant 4.

```
context Port inv: -- Invariant 4
  self.IsAcmeRole() implies
     self.owner.IsAcmeConnector() and (self.required->size()=1) and
    (self.provided->size()=1)
```

B.2.5 Systems

An Acme system represents a graph of interacting components. The UML's concept of package (with the standard «subsystem» stereotype) represents a set of elements, rather than the structure containing them and is not suitable for defining system-level properties. To avoid such problems we use the constrained component stereotype `<<AcmeSystem>>`, with the constraints presented in listing B.5.

Listing B.5: Invariants 5, 6, and 7.

```
context Component inv: -- Invariant 5
  self.IsAcmeSystem() implies
    self.contents()->select(el|
      el.IsKindOf(Component))->asSet()->forAll(comp|
        comp.IsAcmeComponent() or comp.IsAcmeConnector())

inv: -- Invariant 6
  self.IsAcmeSystem() implies
    self.contents()->select(el|
      el.IsKindOf(Port))->asSet()->forAll(prt|
        prt.IsAcmePort() or prt.IsAcmeRole() or prt.IsAcmeProperty())

inv: -- Invariant 7
  self.IsAcmeSystem() implies
    self.ownedPort->forAll(ap|
```

```
      ap.IsAcmePort() or ap.IsAcmeRole() or ap.IsAcmeProperty()) and
    self.HasNoOtherInterfaces()
```

B.2.6 Representations

Acme's representations provide the mechanism to add detail to components and connectors. Acme rep-maps are used to show how higher and lower-level representations relate to each other. We will use the features for packaging components of UML 2.0 to express representations. UML provides two wiring elements (in the UML specification, they are referred to as "specialized connectors"): assembly and delegation. The former provides a containment link from the higher level component to its constituent parts, while the latter provides the wiring from higher level provided interfaces to lower level ones, and from lower level required interfaces to higher level ones. A delegation corresponds to Acme's rep-map concept. To ensure components are connected to other components through connectors, we need to constrain all assembly connectors to link ports to roles. The corresponding well-formedness rule is presented in listing B.6.

Listing B.6: Invariant 8.

```
context connector inv: -- Invariant 8
  self.kind = #assembly implies
    self.end->(exists(cp|cp.role.IsAcmePort())
    and exists(cr|cr.role.IsAcmeRole()) )
```

Figure 2.10 depicts the specification of `server`. The wiring between the internal structure of `server` (a system which contains a topology with three components and the connectors among them) and the `server`'s own ports is achieved with the usage of the `<<delegate>>` connectors. Although Acme explicitly uses the concepts of representation and system for defining subsystems, we make them implicit in our mapping. Making them explicit would not improve the expressiveness of the resulting design and would clutter the diagram by creating an extra level of indirection.

B.2.7 Properties

Properties represent semantic information about a system and its architectural elements. To allow automatic reasoning on them, using OCL, we can make these properties available outside the component's internal scope. Ports can be typed with a provided interface that allows the component user to access its properties. The downsides of representing Acme properties as UML ports are that by doing so we are cluttering the design and extending the interfaces provided by the design element. An `<<AcmeProperty>>` port owns a single provided interface that must provide get and set operations for the property's value and type.

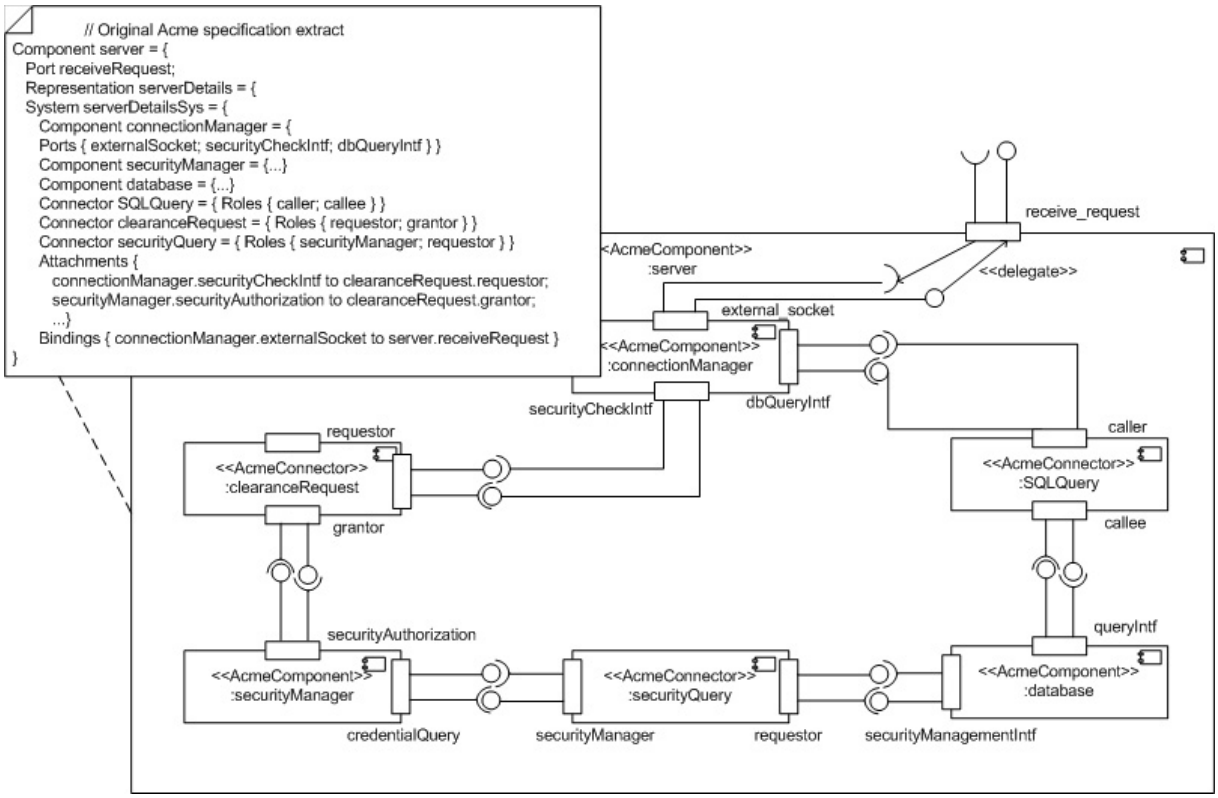


Figure B.2: Detailing a component specification

Listing B.7: Invariant 9.

```
context Port inv: -- invariant 9
  self.IsAcmeProvided() implies
    (self.required->IsEmpty()) and
    (self.provided->size()=1)
```

B.2.8 Constraints (invariants and heuristics)

Constraints allow the specification of claims on how the architecture and its components are supposed to behave. While invariants are conditions that must hold at all times, heuristics are constraints that should hold, although breaking them is possible. In UML, we can express design constraints through OCL. These constraints can be pre-conditions, post-conditions or invariants. Acme's notion of invariant can be directly mapped to its OCL counterpart. However, there is no direct UML semantic equivalent for the notion of heuristic. This could be circumvented by creating the `<<AcmeConstraint>>` stereotype as a specialization of the UML `Constraint` metaclass. The former would have an enumerated attribute with two allowed values: `invariant` and `heuristic`.

B.2.9 Styles and types

An architectural style defines a vocabulary of design elements and the rules for composing them. It is used in the description of families of architectures. Since we have created stereotypes for the several UML constructs used in this Acme to UML mapping, we can now specify architectural styles using these stereotyped elements.

Figure 3 represents the pipe and filter family, an architectural style that defines two types of components, `FilterT` and `UnixFilterT`, a specialization of `FilterT`. The architectural style is defined by means of a UML package, as the family definition does not prescribe a particular topology. It does, however, establish an invariant that states that all the connectors used in a pipe and filter system must conform to `PipeT`.

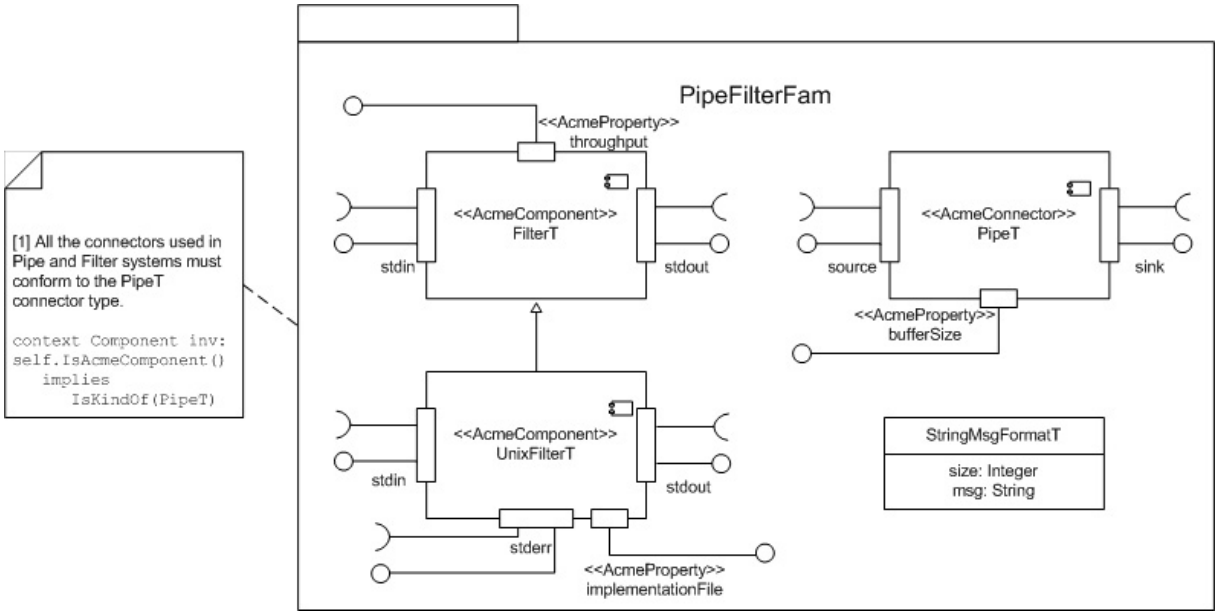


Figure B.3: The pipe and filter family

B.3 Discussion

The presented mapping from Acme to UML is more straightforward than previous approaches. This mainly results from the increased expressiveness provided by the new UML 2.0 design elements. From a structural viewpoint, representing a topology is fairly simple when using UML. This is mainly due to the relative closeness of the sort of structural information that we want to express both at the architectural and design levels. In both cases we have to identify components and the connections among them, possibly at different levels of abstraction.

However, while a connector is regarded as a first class design element by the architecture community, it has no direct mapping in UML 2.0. Our proposal is to promote connectors to first class design elements, by representing them as stereotyped components. This seems to be a sensible option, considering that the evolution of CBD

should provide us with an increasing number of off-the-shelf components and that, the complexity of building component-based software is shifting to the production of glue code. Representing connectors as stereotyped components gives us the extra flexibility to meet this challenge.

The representation of properties is not an easy nut to crack. Perhaps they could be more suitably defined at the meta-level, rather than using the `<<AcmeProperty>>` ports for this purpose, but this still requires further research.

Heuristics are also complex to map directly to UML, as UML provides no direct representation for this concept, although we can use OCL to deal with this problem.

Since Acme does not provide a direct support for component dynamics specification, in this appendix we do not address it. Nevertheless, we could use properties to annotate the architectural entities with information on their expected behavior. For instance, a connector may have a property specifying its protocol with some formalism (e.g. Wright). We could use UML's behavioral modeling features similarly, thus complementing the structural information in the mapped specification with a behavioral specification of the design elements used.

B.4 Related work

A number of mappings among the concepts expressed in ADLs and their representation with UML have been attempted. A possible strategy is to use UML "as is", in the mapping. In [Medvidovic 02], UML is used to express C2 models. In [Garlan 00a], Garlan presents several UML metamodel elements as valid options to express each of the structural constructs defined in Acme. Each mapping becomes the best candidate depending on the goals of the translation from Acme to UML. The semantic mismatch between the ADL and UML concepts is the main drawback of this strategy.

An alternative is to modify the UML metamodel, to increase the semantic accuracy of the mapping [Selic 02]. Unfortunately, this drives us away from the standard, and consequently sacrifices existing tool support.

An interesting compromise is to use UML's extension mechanisms to mitigate conceptual mismatches, while maintaining compatibility with the standard metamodel. Examples of this strategy can be found in [Egyed 01] (C2SADEL to UML), [Cheng 01] (Acme to UML-RT), and [Robbins 98] (C2 and Wright to UML). The latter uses OCL constraints on the metamodel elements which is close to the one proposed in this appendix, but requires a mapping for each ADL and uses an older and notably less expressive version of UML). The approach discussed in this appendix bridges the gap between software architecture and design using an OO modeling notation. All the mappings discussed so far in this section were performed with UML 1.x, whereas here we use the new UML 2.0 metamodel elements, which enhance the language's suitability for component-based design.

The mapping presented in this chapter can be used as a way of assessing the expressiveness of UML 2.0 with respect to representing structural information conveyed by architectural description languages. This kind of exercise has also been performed for other component models discussed in this dissertation. Polák created mappings from SOFA and Fractal to UML 2.0 [Polák 05]. His objective was to start from a specification in SOFA, or Fractal, transform it to the corresponding UML specification, and then use the latter to generate source code. Rather than specifying the well-formedness rules in OCL, Polák hard coded those constraints in the implementation of a prototype to perform the transformation, due to limitations with the UML tools he had available. This problem could have been avoided with more recent tools, such as MagicDraw[2], by using the support for modeling Domain Specific Languages, which includes the ability to define and validate OCL constraints.

Following the publication of our mapping between Acme and UML 2.0, Roh *et al.* proposed an alternative to our connector representation [Roh 04]. Their alternative definition builds on the fact that UML 2.0 connectors are instances typed by the association between the classifiers that they connect. Their metamodel extension is considerably more complex than ours. It requires four stereotyped meta-classes: `ArchRole`, defined as a stereotyped version of the UML 2.0 `ConnectableElement` meta-class is used to represent the roles in the ends of the connector; `ArchComposition`, defined as a stereotyped version of the UML 2.0 `Collaboration` meta-class, allows representing composition patterns among components; `ArchConnector` is a specialization of the stereotype `ArchComposition`, and is used when we want to model connectors that implement a complex protocol. For simple connections, the normal UML connector is used. The main advantage is that this combination of stereotypes allows representing Acme connectors through UML connectors, rather than as components.

Another interesting feature of Roh *et al.*'s proposal is that they use a layered structure for defining their profile. They define a profile for a generic ADL in the M2 meta-level that may be extended by domain-specific ADLs, also at the M2 meta-level. At the M1 meta-level, they expect domain engineers to provide reusable elements of the domain as a framework architecture. When defining an architecture, one can not only apply the domain specific architecture, but also import reusable elements from the framework. This approach mitigates the problem created by the diversity of ADLs, by postponing the definition of constructs which are specific to a given ADL to the domain-specific extensions to the generic ADL.

B.5 Conclusions

We have shown the feasibility of expressing architectural information expressed in Acme using the UML 2.0. It is possible to obtain a mapping from a given ADL to UML,

[2] `http://www.magicdraw.com/`

through a two-step approach. We could first map the architecture from the original ADL to Acme and then use the mapping proposed in this appendix to obtain the corresponding specification in UML. Details lost in the ADL to Acme conversion can always be added later to the resulting UML specification. The proposed mapping builds upon the added expressiveness of UML 2.0 for architectural concepts, when compared to UML's previous versions. The availability of components with ports typed by provided and required interfaces has proved to be a step forward in the exercise of bridging the gap between architectural and design information. This improves traceability between architectural description and its implementation, using the design as a middle layer between them. This traceability is relevant for keeping the consistency between the architecture, design and implementation of a software system. The proposed mapping focuses mainly on structural aspects and design constraints. Although it also points out to ways of dealing with the definition of system properties, including semantics and behavioral specification, further research is required to provide more specific guidance on these aspects.

[This page was intentionally left blank]

Appendix C

Tool support

Contents

Background: *The metrics collection activities described throughout this dissertation use a set of tools developed specifically to support Ontology-Driven Measurement (ODM).*

Objectives: *In this appendix, our goal is to briefly describe the architecture of the tool support built for collecting and analyzing metrics in this dissertation.*

Techniques: *We will use a combination of views to support our discussion on the chosen architecture.*

Results: *The generic architecture presented here can be easily mapped to the specific architecture in each of the experimental works discussed in this dissertation.*

Limitations: *Although it is possible to identify a family of architectures used in this dissertation, each of the instances of this architecture was built separately. This could be improved by following a software product line approach, in the future.*

Conclusions: *We were able to consistently use a family of architectures to support the experimental work described in this dissertation. Among other characteristics, the usage of standard techniques made these implementations interesting prototypes that help understanding how this tool support could evolve toward the integration in current IDEs.*

C.1 Documentation roadmap

This appendix describes the basic architecture for collecting metrics, specifying the expected inputs and outputs of the interacting tools. The appendix is organized as follows: In section C.2 we present an overview of the desired system. We outline the system's main requirements in section C.3. We present the description of the architecture through two different views: a structure view, in section C.4.1, and a dynamic view, in section C.4.2. The mapping between these views is presented in section C.5. We provide a brief discussion on the rationale behind this architecture in section C.6. Finally, we map the architecture to our requirements, in section C.7.

C.2 System overview

The architecture presented in this appendix supports the specification and usage of quantitative metrics in the scope of Experimental Software Engineering (ESE) activities. This support is underpinned by the Ontology-Driven Measurement (ODM) approach. With some variation points, which we will identify in the architecture's description, this generic architecture was applied in our metrics collection and analysis, throughout the work described in this dissertation.

Figure C.1 represents a high level context view of the whole system through a UML use case diagram. The `Experimenter` uses the tool support to conduct three main tasks: `Configure data collection`, `Gather sample`, and `Perform analysis`.

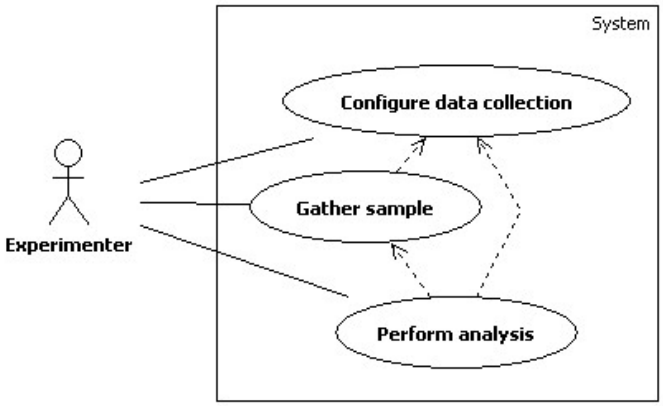


Figure C.1: Context view of the system

`Configure data collection` includes creating configuration files for the used components, such as an Ontology specification and an OCL metrics specification. `Gather sample` consists in depositing in the experiment's repository the raw data for experimentation and using the appropriate component `InstancesGenerator` to represent that raw data as an instance of the chosen ontology. Note that it is beyond of the scope of the tool support described in this section to help in the raw data collection.

Finally, `Perform analysis` consists in computing metrics and testing heuristics on the ontology instantiation, and then using those results to perform the statistical analysis upon them. The interpretation and packaging of results is out of the scope of the tool support provided by the architecture described in this appendix.

C.3 Requirements

The architecture described in this appendix supports the following activities:

- Ontology definition in UML
- Metrics and heuristics definition, in OCL, using the ODM approach
- Representation of the experimental data as an instantiation of the ontology
- Automatic metrics collection and heuristics test
- Automatic statistical analysis of results

To the best of our knowledge, no single tool supports all these activities. Therefore, we need to use a combination of off-the-shelf components with some custom-made components created by us. This combination of components from various sources makes their interoperability an important requirement. In order to facilitate the development of the required glue code, it is important to use tools that allow importing and exporting information in textual format with a well specified grammar.

Concerning distribution, the architecture for our prototypical tool support for experimentation can be run on a single machine. With respect to performance, this is not a critical requirement, either. For the purposes of this dissertation, having a metrics computation, or a statistical analysis, ready in a few seconds, or minutes, is not a critical requirement, as well.

C.4 Views

We present two complementary views of our architecture: a structural, and a dynamic view. The former illustrates the overall configuration of the components used to provide the tool support for our experiments, while the latter makes more explicit the process followed in the metrics definition and collection. We choose to present these views because they are the ones that best represent the main requirements of the tool support.

Other views, such as a detailed implementation view, or a view of the physical view of the architecture would not bring much added value for the discussion of this architecture. The implementation of our custom-made components was not severely constrained by the architecture. Those components had to process text files, transform

them, and produce other text files, with no real-time, or concurrency constraints. Some of those text files were XML files, so the main constraint was that the chosen implementation language would have good support for working with XML. Programming languages such as Java, or C♯ were used and combined (e.g. the `InstancesGenerator` can be defined in C♯ and integrated with an architecture where the `Convert2StatsTool` is developed in Java), for any of the experiments. All communication between components was established through text files. Concerning the physical representation, presenting its detailed view is not particularly useful, either, because all the components can run on a single machine.

C.4.1 Structural view

Primary presentation

The structural view of our architecture, presented in figure C.2, uses a UML 2.0 component diagram. The chosen architecture follows the pipe and filter architectural style (see, for instance, [Garlan 00b]). We identify 5 components (the filters) which communicate through the operating system's file system (the pipe). All the connectors are annotated with the stereotype `<<file system>>`. This denotes that the component playing the `source` role in a connector makes a file available to the file system, so that the component playing the `sink` role can read that file. The artifacts presented in the diagram correspond to the files being produced and consumed by the components. See the element catalog for further details.

Element catalog

We can identify three main types of elements in this view: components, connectors, and artifacts. We use the following components:

- `Repository` - This component is responsible for storing all the experimental data and configuration files. It typically corresponds to the file system where all the necessary files (including sample data, ontology description and other configuration files) are stored.

- `InstancesGenerator` - The raw sample data (`original subjects`) has to be represented as an instantiation of the chosen ontology (`subjects ontology`). This component is responsible for parsing the `original subjects` and creating such an instantiation (the `subjects` artifact).

- `OCL tool` - This component is a UML tool with support to OCL specification and evaluation. In all our experiments, the chosen tool was USE. Although USE only supports a subset of UML 1.*, this subset covers our ontology modeling requirements. USE has both a textual and a graphical mode. For our purposes, we

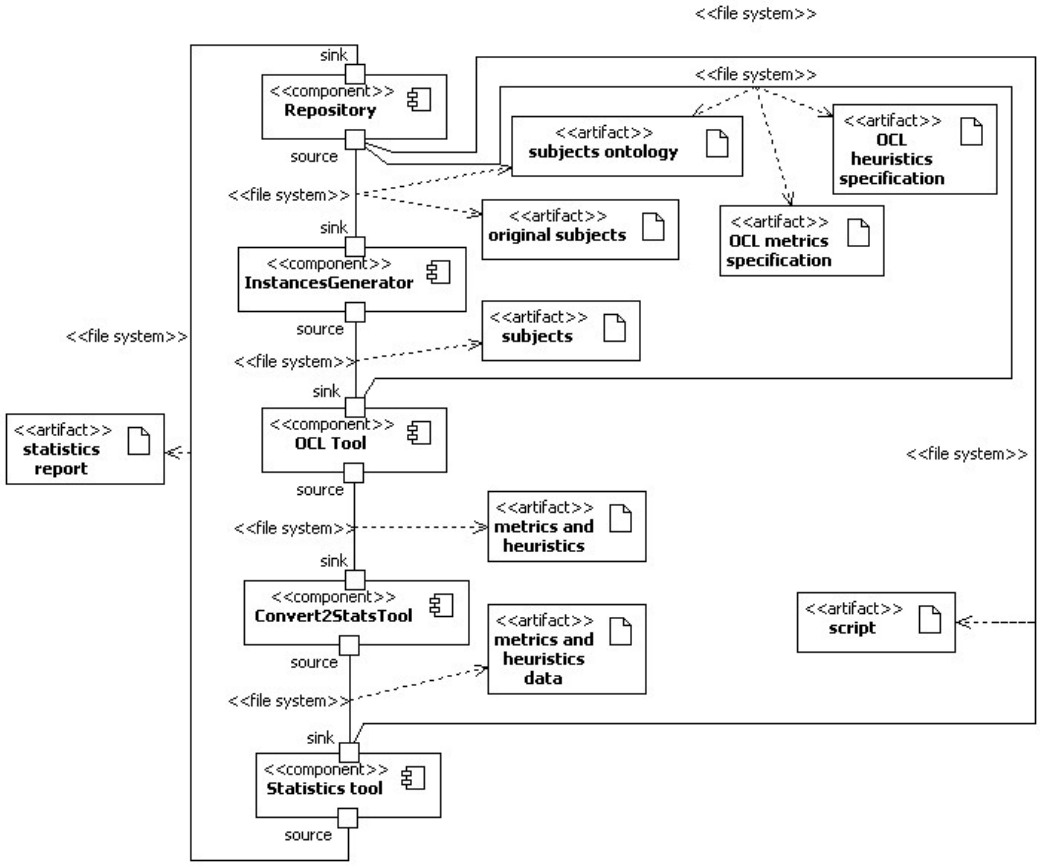


Figure C.2: Structural view

only use the textual mode. USE includes commands for importing model files, where we can specify a model (our ontology), as well as OCL rules for that model. These OCL rules are then used to specify metrics and heuristics for the ontology. The USE tool also includes commands for importing text files with instantiation instructions, as well as queries to the model and its instances. The queries are expressed in OCL. The combination of these four sources (an ontology, its metrics and heuristics specifications, and instantiation instructions) is then used to compute the metrics and test the heuristics. The results produced by the USE tool are finally exported as a text file (using the comma separated values notation).

- `Convert2StatsTool` - The comma separated values (CSV) file has to be translated to a format readable by the chosen `StatisticsTool`. The `Convert2StatsTool` is responsible for this transformation.

- `StatisticsTool` - This component is a statistics tool. In all our experimental works, we used the SPSS tool. Among other features, such as a wide variety of available statistical tests, SPSS includes a production facility which is very useful for automating statistical tests. The production facility uses a textual scripting language of commands that can be used to prepare data and then execute all the necessary statistical tests.

The connectors represented in the architecture are always similar and correspond to the file system where the component playing the `source` role deposits a file (an `artifact`) that is then used by the component playing the `sink` role.

Finally, we consider the following artifacts:

- `original subjects` - This artifact represents the raw data of our experimental work. This data can be stored in any format. It is up to the `InstancesGenerator` to transform the raw data into instances of an ontology. For example, in chapter 4, `original subjects` corresponds to the source code of the JavaBeans. In chapter 5, it corresponds to UML 2.0 components and CCM 3.0 components. In chapter 6, it corresponds to the inspection reports submitted by each inspection team. In chapter 7, it corresponds to a folder with the Eclipse plug-ins.

- `subjects ontology` - This artifact represents the ontology used in each of the experiments. These ontologies were expressed as UML models (most of which were metamodels). These ontologies were expressed in a format readable by the chosen OCL tool (USE).

- `OCL metrics specification` - These metrics are defined in OCL upon the `subjects ontology`. Several examples of such definitions can be found in chapters 4, 5, 6 and 7.

- `OCL heuristics specification` - These heuristics are defined in OCL upon the `subjects ontology`. Examples of heuristics definitions can be found in chapter 4.

- `subjects` - This is the file (or set of files) representing the sample as instances of the `subjects ontology`. These instantiation files are expressed in USE's snapshot specification language.

- `metrics and heuristics` - This artifact contains the results of the OCL queries that compute the metrics. This is a plain text file, with comma separated values.

- `metrics and heuristics data` - The OCL tool used in the computation of metrics and heuristics tests is not very flexible with respect to the output format. So, those raw results are formatted into this artifact, to facilitate their import in a statistics tool.

- `script` - This artifact contains the commands for the statistics tool. In our experimental work this tool was always SPSS, so this artifact is a SPSS command file, for using with the SPSS production facility.

- `statistics report` - This is the output report produced with the statistics tool. SPSS has a proprietary report format that we used without changes. It would be possible to export the results in other formats, such as html, with embedded

images for representing graphical outputs, but this was not necessary for our purposes.

Variability guide

The architecture was specified so that the components are as loosely coupled as possible. In principle, any of these elements could be changed by another component providing a similar functionality. In practice, with the exception of version upgrades, for all the instantiations of this architecture that we have developed, there were always three components that were fixed, and two that changed from one experiment to the next. More generally, these are the most likely points of variation in our architecture for future experiments, as well.

The fixed elements of our architecture are the `Repository`, the `OCL Tool`, and the `Statistics tool`. The `Repository` was the file system. The `OCL tool` used in all experiments was the USE tool. Finally, the `Statistics tool` was SPSS. The variation points were the `InstancesGenerator` and the `Convert2StatsTool`.

The `InstancesGenerator` is the component of our architecture that is responsible for generating an instantiation of the chosen ontology that represents the subject's sample. As such, this is the most variable element of the architecture. Depending on what our subjects are, the ontology specifying them may vary. Each ontology requires its own `InstancesGenerator`. As we have used a different ontology for each of our experimental works, this implied using a different instances generator.

The `Convert2StatsTool` is essentially a glue component. In each of our experimental works, the set of metrics being extracted from the sample was different, and this implied that the format of the results would also differ, from one experiment to the next. These results had to be converted into a data format that SPSS could read, and this conversion was the task of the component that would assume the role of `Convert2StatsTool` in each architecture instantiation.

Related views

The dynamic view, presented in section C.4.2, illustrates the activities that, together, represent the metrics collection process supported by this architecture.

C.4.2 Dynamic view

Primary presentation

Figure C.3 represents an activity diagram in UML 2.0, decorated with swim lanes, to illustrate the responsibilities of each of each of the used components, and those activities which require direct intervention by the `Experimenter`. The experimenter is responsible for creating a set of configuration files, including a `subjects ontology`, an

`OCL metrics specification`, an `OCL heuristics specification`, and a `script` with the commands for the statistical analysis. The experimenter is also responsible for gathering a sample of subjects that will be tested in the statistical analysis. Finally, the experimenter is also responsible for interpreting and packaging the results. In this diagram, the activities' description is decorated with object flows, where the objects are the files being written during an activity (this is denoted by the `<<write>>` stereotype) and being read during a subsequent activity (this is denoted by the `<<read>>` stereotype).

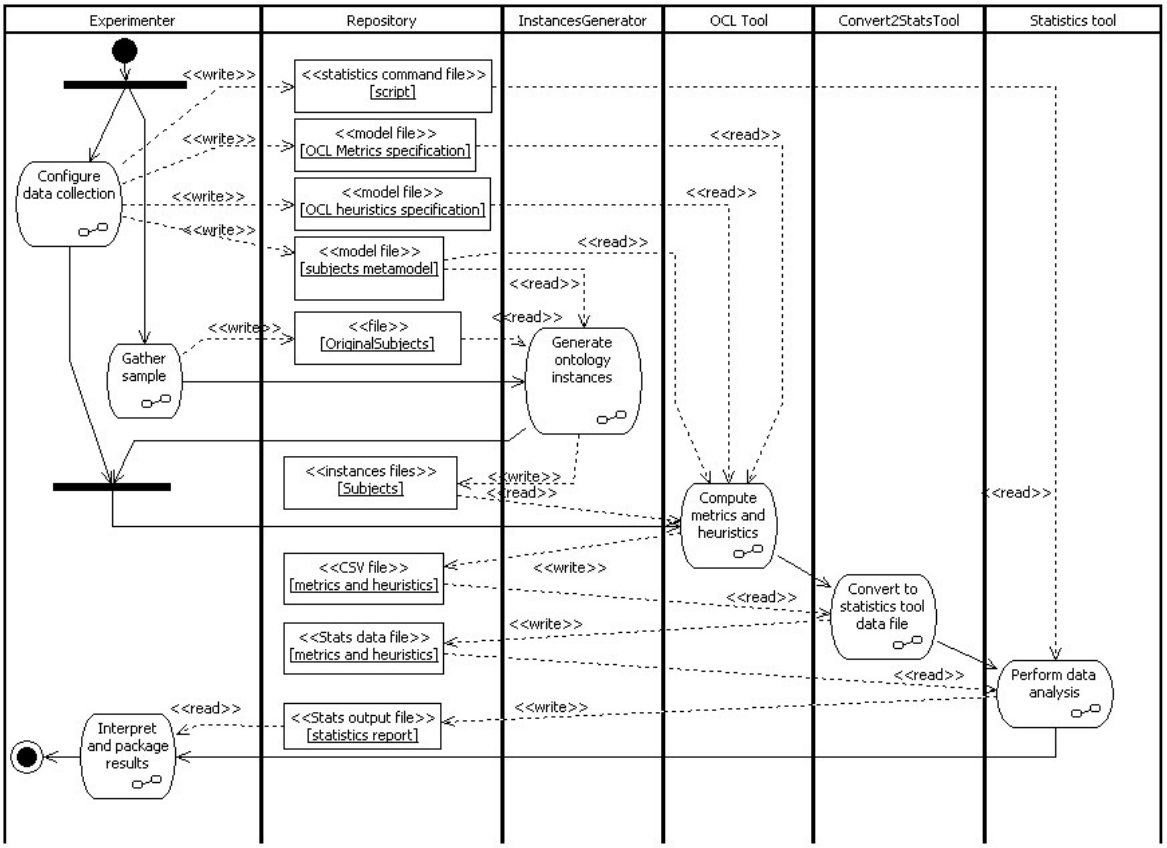


Figure C.3: Metrics collection activities

Element catalog

In this catalog, we have essentially three sorts of elements: activities, files and swim lanes. We also have transitions among these elements. A transition between activities denotes the sequence in which the activities are performed. The dotted arrows from an activity to a file denote that the activity produces that file. The dotted arrows from a file to an activity denotes that the activity consumes that file. The files correspond to the artifacts discussed in section C.4.1, so please refer to the data catalog in that session for further information on them. The swim lanes correspond to the components, also discussed in section C.4.1. The exception is the swim lane dedicated to the `Experimenter`, a swim lane used for expressing the activities conducted by the human experimenter

and how they fit into the overall architecture. Note that the experimenter's activities are conducted out of the boundaries of the tool support. We represent them here anyway, to provide a better context for the discussion on the overall process.

We now describe the activities in further detail:

- `Configure data collection` - This activity is carried out by the experimenter and consists on preparing the whole experimental environment. This involves specifying the `subjects ontology`, as well as OCL metrics and heuristics. It also involves specifying the statistics tests to be performed upon the sample, during the experiment.

- `Gather sample` - This activity corresponds to the data collection part of the experimental process. Its outcome is a set of files with the raw data for experimentation.

- `Generate ontology instances` - This activity corresponds to expressing the raw experimental data in terms of instances of the chosen ontology. The activity is performed by the `InstancesGenerator`, and generates a file (or set of files) representing those instances.

- `Compute metrics and heuristics` - The instances generated in the previous activity are used in this activity to compute metrics and test heuristics upon the instances and their corresponding metrics' values.

- `Perform data analysis` - This activity consists in performing the data analysis and statistics tests to the hypotheses being researched. It can be automated, to ensure its repeatability, by using a command script.

- `Interpret and package results` - Finally, this activity is performed by the experimenter, using the results obtained during data analysis. The results of this activity are typically written down in an experimental report, for dissemination among interested stakeholders, but the details of these interpretation and packaging operations fall beyond the scope of the tool support described here.

Variability guide

The variability in the dynamic matches that of the structural view. From an architectural point of view, the activities in the swim lanes `Repository`, `OCL tool`, and `Statistics tool` remain the same, from one experiment to the next. They are configured by different files, handle different data, compute different metrics and heuristics, and perform different statistics tests, but the components performing those tasks remain the same. The points of variation in this view are the activities in the `Experimenter`, `InstancesGenerator`, and the `Convert2StatsTool` swim lanes. We have

already discussed the inherent variability of the latter two while presenting the variability guide for the static structure.

We will now focus on the `Experimenter` swim lane. It is not surprising that the activities of the experimenter, which correspond to how this architecture is used, in practice, are the main point of variability in this dynamic view. However, we do not consider these variations to be within the scope of the architecture described here. The kind of tasks performed by the `Experimenter` remains essentially the same, from one experiment to the next. Each experimental design feature will lead to a specific impact in these tasks. Concerning the configuration, the nature of the sample conditions the adoption of the ontology: in some cases, the experimenter will adopt an existing ontology, while in other the experimenter has to adapt an existing ontology, or create a new one from scratch. With respect to the sample gathering, this too, varies with the nature of the sample. The raw data must be collected in a format suitable for automated ontology instance generation. However, all these variations, as well as those concerning data interpretation and packaging, are beyond the scope of the architecture presented in this appendix.

Related views

The structural view, presented in section C.4.1, illustrates the components that support the activities described in this dynamic view.

C.5 Mapping between the views

The mapping between the structural and dynamic views is relatively straightforward, as we have used the same identifiers in both views:

- Both views represent the same model. Each component in the structural view has a corresponding swim lane in the dynamic view, which allows specifying which are the activities performed by each of the participant components.

- The only swim lane in the dynamic view which does not correspond to a component in the structural view is the swim lane of the experimenter.

- All the artifacts in the structural view are represented, with the same name, in the dynamic view, as object flows. While in the structural view we use a generic `<<artifact>>` stereotype, in the dynamic view we use a more specific stereotype, to differentiate among the different types of files being written and read by the activities. In both views, they correspond to files to be stored in the file system of the machine running this tool support.

C.6 Architecture Analysis and Rationale

The development of the prototypical implementation of the tool support for our experiments aimed at reusing available existing components, where possible, combined with custom-made components for the remaining functionalities. It was important that our reused components would be as independent as possible from the domain under scrutiny in each experimental work, so that the architecture would be flexible.

The option for using a style which is, in essence, a pipe and filter, using the file system as a connector and the set of custom-made and off-the-shelf components as filters, was mostly constrained by the available resources. We developed all the custom made components. With only one developer, options such as extending existing tools, or integrating several tools into a new one (e.g. a plug-in for an Integrated Development Environment) would divert resources from the experimentation *per se*, which was our core concern.

Although it would be possible to devise a distributed version of this architecture, for the purposes of this dissertation, this would not bring any relevant benefits. Distribution might make sense in an environment where several installations of this system would be required. For instance, we could be interested in making the statistics component available on a remote server, if the cost of the distributed solution was lower than the licensing costs involved in using several copies of the same statistics tool.

The choice of the off-the shelf components used in the implementations of this architecture was also driven by a combination of flexibility of those components to support the diversity of experiments we wanted to conduct and the availability of those components in our context. Both the USE tool and SPSS are flexible, in the sense that we can script their usage in plain text files.

The USE tool allows scripting the loading of a particular ontology and its instantiation, as well as the definition and computation of OCL rules upon the instances of that ontology. Although the USE tool is a research prototype and this has some efficiency drawbacks when dealing with large samples, few (if any) UML tools available when we started building this architecture supported OCL evaluation, scripted instantiation of models, and were open source, freely available, and as stable as USE. These combination of characteristics made USE a good test bed for the ODM approach. As OCL support is becoming generalized in modern UML tools, replacing USE by a more modern tool that provides the key features (ontology definition, support for its instantiation through scripts, and full OCL support) is an option to consider, in the future.

With respect to SPSS, this is a mature and widely used commercial statistics package. Among other characteristics, such as an adequate robustness, usability, and full coverage of the statistics tests we used in this dissertation, our long experience with the tool from other research projects, and the scripting ability of the tool made it a good candidate for our architecture. The scripting ability allows repeating exactly the same

process with different samples, which is an important characteristic for making experiments replicable. Although other competing statistics packages with similar features might have been used, as well, factors such as the availability, suitability for the task, and our prior experience with this one were decisive in our option.

C.7 Mapping architecture to requirements

C.7.1 Ontology definition in UML

This requirement is supported by the `OCL tool` component. The OCL tool is, of course, a UML tool with OCL support. As such, one can define the ontology as a UML model using the same tool which will be used for defining and collecting metrics and heuristics in OCL. An alternative is to use another UML tool for defining the ontology and then exporting the ontology to the UML tool with support to OCL.

C.7.2 Metrics and heuristics definition, in OCL, using the ODM approach

This requirement is supported by the `OCL tool` component. The metrics are defined as OCL rules which are added to the ontology (i.e. to a UML model, or metamodel).

C.7.3 Representation of the experimental data as an instantiation of the ontology

This requirement is supported by the `InstancesGenerator` tool. The sole purpose of this component is to parse the existing raw data sample and translate it into an instantiation of the chosen ontology, that can then be used by the OCL tool.

C.7.4 Automatic metrics collection and heuristics test

This requirement is supported by the `OCL tool`. As discussed in this dissertation, the OCL expressions are an executable specification of the metrics and heuristics. As such, the `OCL tool` can be used to compute the metrics and to test if any heuristics are being violated.

C.7.5 Automatic statistical analysis of results

This requirement is supported by the `Statistics tool` component. The data for statistical analysis is pre-processed by the `Convert2StatsTool` component, so that it is transformed into a format suitable for being imported by the statistics tool.

Bibliography

[Abran 04] Alain Abran, James W. Moore, Pierre Bourque & Robert Dupuis, editors. Guide to the Software Engineering Body of Knowledge (SWEBOK). IEEE Computer Society, 2004.

[Abreu 94a] Fernando Brito Abreu & Rogério Carapuça. *Candidate Metrics for Object-Oriented Software within a Taxonomy Framework*. Journal of Systems and Software, vol. 26, no. 1, pages 87–96, 1994.

[Abreu 94b] Fernando Brito Abreu & Rogério Carapuça. *Object-Oriented Software Engineering: Measuring and Controlling the Development Process*. In 4th International Conference on Software Quality (ICSQ'94), McLean, Virginia, USA. American Society for Quality, 1994.

[Abreu 96] Fernando Brito Abreu & Walcélio Melo. *Evaluating the Impact of Object-Oriented Design on Software Quality*. In 3rd International Software Metrics Symposium (Metrics'96), Berlin, Germany. IEEE Computer Society, 1996.

[Abreu 99] Fernando Brito Abreu, Luís Miguel Ochoa & Miguel Goulão. *The GOODLY Design Language for MOOD2 Metrics Collection*. In Fernando Brito e Abreu, Houari Sarahoui & Horst Zuse, editors, 3rd ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'1999), Lisbon, Portugal. 1999.

[Abreu 01a] Fernando Brito Abreu. *Engenharia de Software Orientado a Objectos: uma Aproximação Quantitativa*. Phd, Instituto Superior Técnico, Universidade Técnica de Lisboa, 2001.

[Abreu 01b] Fernando Brito Abreu. *Using OCL to formalize object oriented metrics definitions*. Technical Report ES007/2001, INESC, May 2001.

[Albrecht 83] Allan J. Albrecht & John E. Gaffney. *Software Function, Source Lines of Code and Development Effort Prediction: A Software Sci-*

ence Validation. IEEE Transactions on Software Engineering, vol. 9, no. 6, pages 639–648, 1983.

[Aldrich 08] Jonathan Aldrich. *Using Types to Enforce Architectural Structure*. In 7th Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), pages 211–220, Vancouver, Canada. IEEE Computer Society, 2008.

[Allen 97] Robert Allen & David Garlan. *A Formal Basis for Architectural Connection*. ACM Transactions on Software Engineering and Methodology, vol. 6, pages 213–249, 1997.

[Alvaro 05] Alexandre Alvaro & Silvio Romero de Lemos Meira. *Software component certification: a survey*. In 31st EUROMICRO Conference on Software Engineering and Advanced Applications, pages 106–113, Porto, Portugal. IEEE Computer Society, 2005.

[Aoyama 98] Mikio Aoyama. *New Age of Software Development: How Component-Based Software Engineering Changes the Way of Software Development?* In International Conference on CBSE, 1998.

[Atkinson 01] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst & Jörg Zettel. Component-based Product-Line Engineering with UML. The Addison-Wesley Object Technology Series. Addison-Wesley Publishing Company, 2001.

[Atkinson 05] Colin Atkinson & Oliver Hummel. *Towards a Methodology for Component-Driven Design*. In Nicholas Guelfi, editor, First International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2004), volume LNCS 3475, pages 23–33, Kirchberg, Luxembourg. Springer, 2005.

[Bachman 00] Felix Bachman, Len Bass, Charles Buhman, Santiago Cornella-Dorda, Fred Long, John Robert, Robert Seacord & Kurt Wallnau. *Volume II: Technical Concepts of Component-Based Software Engineering*. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, May 2000.

[Barbacci 98] Mario R. Barbacci & Charles B. Weinstock. *Mapping MetaH into ACME*. Technical Report CMU/SEI-98-SR-006, Software Engineering Institute, July 1998.

[Baroni 02a] Aline Lúcia Baroni. *Formal Definition of Object-Oriented Design Metrics*. Msc, Vrije Universiteit Brussel - Belgium, in collaboration with École des Mines de Nantes - France and Universidade Nova de Lisboa - Portugal, 2002.

[Baroni 02b] Aline Lúcia Baroni & Fernando Brito Abreu. *Formalizing Object-Oriented Design Metrics upon the UML Meta-Model*. In Brazilian Symposium on Software Engineering, Gramado - RS, Brazil. 2002.

[Baroni 03] Aline Lúcia Baroni & Fernando Brito Abreu. *A Formal Library for Aiding Metrics Extraction*. In 4th International Workshop on Object-Oriented Reengineering (WOOR2003) at ECOOP'2003, Darmstadt, Germany. 2003.

[Baroni 05a] Aline Lúcia Baroni, Fernando Brito Abreu & Coral Calero. *Finding Where to Apply Object-Relational Database Schema Refactorings: an Ontology-Guided Approach*. In X Jornadas sobre Ingeniería del Software y Bases de Datos (JISBD 2005), Granada, Spain. 2005.

[Baroni 05b] Aline Lúcia Baroni, Coral Calero, Mario Piattini & Fernando Brito Abreu. *A Formal Definition for Object-Relational Database Metrics*. In 7th International Conference on Enterprise Information System (ICEIS 2005), Miami, USA. 2005.

[Bartezko 01] D. Bartezko, C. Fischer, M. Möller & H. Wehrheim. *Jass - Java with assertions*. Eletronic Notes in Theoretical Computer Science, Proceedings of RV 01, vol. 55, no. 2, 2001.

[Basili 85] Victor R. Basili. *Quantitative Evaluation of Software Engineering Methodology*. In 1st Pan Pacific Computer Conference, Melbourne, Australia. 1985.

[Basili 94] Victor R. Basili, Gianluigi Caldiera & Dieter H. Rombach. *Goal Question Metric Paradigm*. In John J. Marciniak, editor, Encyclopedia of Software Engineering, volume 1, pages 469–476. John Wiley & Sons, 1994.

[Basili 96a] Victor R. Basili. *The role of experimentation in software engineering: past, current, and future*. In 18th International Conference on Software Engineering (ICSE'1996), pages 442–449, Berlin, Germany. IEEE Computer Society, 1996.

[Basili 96b] Victor R. Basili, Scott Green, Oliver Laitenberger, Filippo Lanubile & Forrest Shull. *The Empirical Investigation of Perspective-Based Reading*. Empirical Software Engineering, vol. 1, no. 2, pages 133–164, 1996.

[Bass 01] Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord & Kurt Wallnau. *Volume I: Market Assessment of Component-Based Software Engineering*. Technical Note CMU/SEI-2001-TN-007, Software Engineering Institute, May, 2000 2001.

[Beck 99] Kent Beck & Martin Fowler. *Bad Smells in Code*. In Martin Fowler, editor, Refactoring: improving the design of existing code, Object Technology Series, pages 75–88. Addison Wesley Longman, Inc., 1999.

[Benestad 05] Hans Christian Benestad, Erik Arisholm & Dag I. K. Sjøberg. *How to Recruit Professionals as Subjects in Software Engineering Experiments*. In E. Hustad, B.E. Munkvold, K. Rolland & L.S. Flak, editors, Information Systems Research in Scandinavia (IRIS), Kristiansand, Norway. Department of Information Systems, Agder University College, 2005.

[Berezin 98] Sergey Berezin, Sérgio Campos & Edmund M. Clarke. *Compositional Reasoning in Model Checking*. Lecture Notes in Computer Science, vol. 1536, pages 81–103, 1998.

[Bertoa 02] Manuel Bertoa & Antonio Vallecillo. *Quality Attributes for COTS Components*. In Mario Piattini, Fernando Brito Abreu, Houari Sahraoui & Geert Poels, editors, 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2002), Málaga, Spain. 2002.

[Bertoa 04] Manuel Bertoa & Antonio Vallecillo. *Usability metrics for software components*. In 8th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2004), Oslo, Norway. 2004.

[Bertoa 06] Manuel Bertoa, José Troya & Antonio Vallecillo. *Measuring the usability of software components*. Journal for Systems and Software, vol. 79, no. 3, pages 427–439, 2006.

[Beugnard 99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau & Damien Watkins. *Making Components Contract Aware*. IEEE Computer, vol. 32, no. 7, pages 38–45, 1999.

[Biffl 02] Stefan Biffl & Michael Halling. *Investigating the Influence of Inspector Capability Factors with Four Inspection Techniques on Inspection Performance*. In Eighth IEEE International Symposium on Software Metrics (Metrics'02), 2002.

[Binns 93] Pam Binns & Steve Vestal. *Formal real-time architecture specification and analysis*. In Tenth IEEE Workshop on Real-Time Operating Systems and Software (RTOOS'1993), pages 104–108, New York, USA. IEEE Computer Society, 1993.

[Bisant 89] David B. Bisant & James R. Lyle. *A Two-Person Inspection Method to Improve Programming Productivity*. IEEE Transactions on Software Engineering, vol. 15, no. 10, pages 1294–1304, 1989.

[Boehm 81] Barry W. Boehm. Software Engineering Economics. Prentice-Hall, Englewood Cliffs, NJ, EUA, 1981.

[Bourgeois 96] Karen V. Bourgeois. *Process Insights from a Large-Scale Software Inspections Data Analysis*. CROSSTALK: The Journal of Defense Software Engineering, pages 17–23, 1996.

[Boxall 04] Marcus A. S. Boxall & Saeed Araban. *Interface Metrics for Reusability Analysis of Components*. In Australian Software Engineering Conference (ASWEC'2004), pages 40–51, Melbourne, Australia. IEEE Computer Society, 2004.

[Briand 98] Lionel Briand, Khaled El Emam, Oliver Laitenberger & Thomas Fussbroich. *Using simulation to build inspection efficiency benchmarks for development projects*. In 20th international conference on Software engineering (ICSE'1998), pages 340 – 349, Kyoto, Japan. IEEE Computer Society, 1998.

[Brooke 02] Chris Brooke. *The Return on Investment on Commercial off-the-shelf (COTS) Software Components - Preliminary Study Results*. White paper, Component Source, August 2002.

[Brooks 97] Andy Brooks. *Meta Analysis -A Silver Bullet - for Meta-Analysts*. Empirical Software Engineering, vol. 2, no. 4, pages 333–338, 1997.

[Brownsword 00] L. Brownsword, T Oberndorf & C. A. Sledge. *Developing New Processes for COTS-Based Systems*. IEEE Software, vol. 17, no. 4, pages 48–55, 2000.

[Bruneton 04] E. Bruneton, T. Coupaye & J. B. Stefani. *The Fractal Component Model*. Specification, The ObjectWeb Consortium, February 2004.

[Bryton 07] Sérgio Bryton & Fernando Brito Abreu. *Towards Paradigm-Independent Software Assessment*. In Ricardo Machado, Fernando Brito Abreu & Paulo Rupino Cunha, editors, 6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007), pages 40–51, Lisbon, Portugal. IEEE Computer Society, 2007.

[Bryton 08] Sérgio Bryton. *Modularity Improvements with AOP*. Msc., Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2008.

[Budgen 03] David Budgen & Mitchell Thomson. *CASE tool evaluation: experiences from an empirical study*. Journal of Systems and Software, vol. 67, no. 2, pages 55–75, 2003.

[Bures 06] Tomás Bures, Petr Hnetynka & Frantisek Plásil. *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model*. In Fourth International Conference on Software Engineering Research, Management and Applications (SERA 2006), pages 40–48, Seattle, Washington, USA. IEEE Computer Society, 2006.

[Bures 07] Tomás Bures, Petr Hnetynka & Frantisek Plásil. *Runtime Concepts of Hierarchical Software Components*. International Journal of Computer & Information Science, vol. 8, pages 454–463, 2007.

[Calero 05] Coral Calero, Francisco Ruiz, Aline Lúcia Baroni, Fernando Brito Abreu & Mario Piattini. *An Ontological Approach to Describe the SQL:2003 Object-Relational Features*. Computer Standards and Interfaces, 2005.

[Campbell 05] Donald T. Campbell & Julian C. Stanley. Experimental and Quasi-Experimental Designs for Research. Houghton Mifflin Company. Originally published in N. L. Gage (ed.) Handbook of research on teaching (pp. 1-76), Chicago, Rand McNally, 2005.

[Canal 03] Carlos Canal, Lidia Fuentes, Ernesto Pimentel, José María Troya & Antonio Vallecillo. *Adding roles to CORBA objects*. IEEE Transactions on Software Engineering, vol. 29, no. 3, pages 242–260, 2003.

[Chaki 07] Sagar Chaki, James Ivers, Peter Lee, Kurt Wallnau & Noam Zeilberger. *Model-Driven Construction of Certified Binaries*. In ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007), volume LNCS 4735, pages 666–681, Nashville, USA. Springer, 2007.

[Chang 07] Hervé Chang & Philippe Collet. *Patterns for Integrating and Exploiting Some Non-Functional Properties in Hierarchical Software Components*. In 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07), pages 83–92, 2007.

[Chatfield 84] Christopher Chatfield. The Analysis of Time Series: An Introduction. Chapman and Hall, 3rd edition, 1984.

[Cheng 01] Shang-Wen Cheng & David Garlan. *Mapping Architectural Concepts to UML-RT*. In 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), Las Vegas, Nevada, USA. 2001.

[Cherniavsky 91] John C. Cherniavsky & Carl H. Smith. *On Weyuker's Axioms For Software Complexity Measures*. IEEE Transactions on Software Engineering, vol. 17, no. 6, pages 636–638, 1991.

[Chidamber 94] Shyam R. Chidamber & Chris F. Kemerer. *A Metrics Suite for Object Oriented Design*. IEEE Transactions on Software Engineering, vol. 20, no. 6, pages 476–493, 1994.

[Chrissis 03] Mary Beth Chrissis, Bart Broekman, Sandy Shrum & Mike Konrad. CMMI: Guidelines for Process Integration and Product Improvement. SEI Series on Software Engineering. Addison-Wesley Professional, 2003.

[Cicalese 99] Cynthia Della Torre Cicalese & Shmuel Rotensreich. *Behavioral Specification of Distributed Software Component Interfaces*. IEEE Computer, vol. 32, no. 7, pages 46–53, 1999.

[Coglianese 93] Lou Coglianese & Roy Szymanski. *DSSA-ADAGE: An Environment for Architecture-based Avionics Development*. In Advisory Group for Aeronautical Research and Development (AGARD'93), 1993.

[Cook 76] Thomas D. Cook & Donald T. Campbell. *The design and conduct of quasi-experiments and true experiments in field settings*. In

M. D. Dunette, editor, Handbook of industrial and organizational psychology, pages 223–326. Rand MacNally, Chicago, 1976.

[Councill 01] Bill Councill. *Third-Party Certification and Its Required Elements*. In Ivica Crnkovic, Heinz Schmidt, Judith A. Stafford & Kurt Wallnau, editors, 4th ICSE Workshop on Component-Based Software Engineering (CBSE 2001), Toronto, Canada. IEEE Computer Society, 2001.

[Counsell 07] Steve Counsell, George Loizou & Rajaa Naijar. *Quality of manual data collection in Java software: an empirical investigation*. Empirical Software Engineering, vol. 12, pages 275–293, 2007.

[Creswell 03] John W. Creswell. Research Design: Qualitative, Quantitative and Mixed Methods Approaches. SAGE Publications, 2nd edition, 2003.

[Crnkovic 02] Ivica Crnkovic & Magnus Larsson. Building Reliable Component-Based Software Systems. Artech House Publishers, Boston, 2002.

[Crnkovic 04] Ivica Crnkovic, Heinz Schmidt, Judith A. Stafford & Kurt Wallnau. *6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*. ACM SIGSOFT Software Engineering Notes, vol. 29, no. 3, pages 1–7, 2004.

[Crnkovic 06] Ivica Crnkovic, Stig Larsson & Michel Chaudron. *Component-based Development Process and Component Lifecycle*. In International Conference on Software Engineering Advances (ICSEA'06), Tahiti, French Polynesia. 2006.

[Dashofy 01] Eric M. Dashofy, André van der Hoek & Richard N. Taylor. *A Highly-Extensible, XML-Based Architecture Description Language*. In 2nd Working IEEE/IFIP Conference on Software Architecture (WICSA'2001), pages 103–112, Amsterdam, Netherlands. IEEE Computer Society, 2001.

[DeMichiel 06] Linda DeMichiel & Michael Keith. *JSR 220: Enterprise JavaBeans, version 3.0*. Technical report, Sun Microsystems, May 2006.

[Deming 00] W. Edwards Deming. Out of the Crisis. The MIT Press, Cambridge, MA, EUA, 1st edition, 2000.

[D'Souza 98] Desmond Francis D'Souza & Alan Cameron Wills. Objects, Components and Frameworks with UML: The Catalysis Approach. Addison Wesley Longman, Reading, Massachussets, 1998.

[Dumke 00] Reiner Dumke & Andreas Schmietendorf. *Possibilities of the Description and Evaluation of Software Components*. Metrics News, vol. 5, no. 1, 2000.

[Duncan 98] A. Duncan & U. Hölzle. *Adding contracts to Java with handshake, TRCS98-32*. Technical report, University of California at Santa Barbara, 1998.

[Dybå 05] Tore Dybå, Barbara Ann Kitchenham & Magne Jørgensen. *Evidence-based software engineering for practitioners*. IEEE Software, vol. 22, no. 1, pages 58–65, 2005.

[Dyer 92a] M. Dyer. *Verification based inspection*. In 26th Annual Hawaii International Conference on System Sciences (HICSS 1992), volume 2, pages 418–427, Kauai, HI, USA. IEEE Computer Society, 1992.

[Dyer 92b] Michael Dyer. The Cleanroom Approach to Quality Software Development. Wiley, 1992.

[Egyed 01] Alexander Egyed & Nenad Medvidovic. *Consistent Architectural Refinement and Evolution using the Unified Modeling Language*. In 1st Workshop on Describing Software Architecture with UML, co-located with ICSE 2001, pages 83–87, Toronto, Canada. 2001.

[Estublier 02] Jacky Estublier & Jean-Marie Favre. *Component Models and Technology*. In Ivica Crnkovic & Magnus Larsson, editors, Building Reliable Component-Based Software Systems, pages 57–86. Artech House, 1 edition, 2002.

[Fagan 76] Michael E. Fagan. *Design and Code Inspections to Reduce Errors in Program Development*. IBM Systems Journal, vol. 15, no. 3, pages 182–211, 1976.

[Fagan 86] Michael E. Fagan. *Advances in Software Inspections*. IEEE Transactions on Software Engineering, vol. 12, no. 7, pages 744–753, 1986.

[Fenton 94] Norman Fenton. *Software Measurement: A Necessary Scientific Basis*. IEEE Transactions on Software Engineering, vol. 20, no. 3, pages 199–206, 1994.

[Fenton 02] Norman Fenton, Paul Krause & Martin Neil. *Software Measurement: Uncertainty and Causal Modelling*. IEEE Software, vol. 10, no. 4, pages 116–122, 2002.

[Fenton 06] Norman Fenton, Martin Neil, William Marsh, Peter Hearty, Paul Krause & Rajat Mishra. *Predicting Software Defects in Varying Development Lifecycles using Bayesian Nets*. Information and Software Technology, 2006.

[Fitzgerald 05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat & Marcel Verhoef. Validated Designs for Object-oriented Systems. Springer, New York, 2005.

[Friedman 37] Milton Friedman. *The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance*. Journal of the American Statistical Association, vol. 32, no. 200, pages 675–701, 1937.

[Fukazawa 03] Yoshiaki Fukazawa, Hironori Washizaki, Hirokazu Yamamoto, Takao Adachi, Yuhki Sakai, Kohzo Satoh & Daiki Hoshi. *FukaBeans: JavaBeans Components Library*, `http://www.fuka.info.waseda.ac.jp/Project/CBSE/fukabeans/`, 2003.

[Gamma 95] Eric Gamma, Richard Helm, Ralph Johnson & John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, Reading, MA, USA, 1995.

[Garcia 04] Félix Garcia, Francisco Ruiz, Manuel Bertoa, Coral Calero, Marcela Genero, Luis Olsina, M. Martín, C. Quer, N. Tondori, S. Abrahao, Antonio Vallecillo & Mario Piattini. *Una Ontología de la Médición del Software*. Technical report UCLM DIAB-04-02-2, Universidad de Castilla-La Mancha, February 2004.

[Garlan 93] David Garlan & Mary Shaw. An Introduction to Software Architecture, volume 1. World Scientific Publishing Company, 1993.

[Garlan 94] David Garlan, R. Allen & J. Ockerbloom. *Exploiting style in architectural desing environments*. In The Second ACM Symposium on the Foundations of Software Engineering (SIGSOFT'94), pages 179–185, 1994.

[Garlan 00a] David Garlan & Andrew J. Kompanek. *Reconciling the Needs of Architectural Description with Object-Modeling Notations*. In Andy Evans, Stuart Kent & Bran Selic, editors, «UML» 2000, volume 1939 of *Lecture Notes in Computer Science*, pages 498–512, York, UK. Springer, 2000.

[Garlan 00b] David Garlan, Robert T. Monroe & David Wile. *Acme: Architectural Description of Component-Based Systems*. In Gary T. Leavens & Murali Sitaraman, editors, Foundations of Component Based Systems, pages 47–68. Cambridge University Press, 2000.

[Garlan 03] David Garlan. *Formal Modeling and Analysis of Software Architecture: Components, Connectors and Events*. In Marco Bernardo & Paola Inverardi, editors, Formal Methods for Software Architectures, volume 2804 of *LNCS*, pages 1–24. Springer, Bertinoro, Italy, 2003.

[Gill 03] Nasib. S. Gill & P. S. Grover. *Component-Based Measurement: Few Useful Guidelines*. ACM SIGSOFT Software Engineering Notes, vol. 28, no. 6, pages 4–4, 2003.

[Gill 04] Nasib. S. Gill & P. S. Grover. *Few Important Considerations for Deriving Interface Complexity Metric for Component-Based Software*. ACM SIGSOFT Software Engineering Notes, vol. 29, no. 2, 2004.

[Glass 02] R. L. Glass, I. Vessey & V. Ramesh. *Research in software engineering: an analysis of the literature*. Information and Software Technology, vol. 44, pages 491–506, 2002.

[Gosling 96] James Gosling, Bill Joy & Guy L. Steele. The Java Language Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, 1996.

[Goulão 02a] Miguel Goulão & Fernando Brito Abreu. *The Quest for Software Components Quality*. In 26th International Computer Software and Applications Conference (COMPSAC'2002), Oxford, England. IEEE Computer Society, 2002.

[Goulão 02b] Miguel Goulão & Fernando Brito Abreu. *Towards a Components Quality Model*. In Work in Progress Session of the 28th Euromicro Conference (Euromicro 2002), Dortmund, Germany. 2002.

[Goulão 03] Miguel Goulão & Fernando Brito Abreu. *Bridging the gap between Acme and UML for CBD*. In Specification and Verification of Component-Based Systems (SAVCBS'2003), at the ESEC/FSE'2003, Helsinki, Finland. 2003.

[Goulão 04a] Miguel Goulão & Fernando Brito Abreu. *Cross-Validation of a Component Metrics Suite*. In IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2004), Malaga, Spain. 2004.

[Goulão 04b] Miguel Goulão & Fernando Brito Abreu. *Formalizing Metrics for COTS*. In Eric Dubois & Xavier Franch, editors, International Workshop on Models and Processess for the Evaluation of COTS Components (MPEC 2004) at ICSE 2004, pages 37–40, Edimburgh, Scotland. IEE, 2004.

[Goulão 04c] Miguel Goulão & Fernando Brito Abreu. *Independent validation of a component metrics suite*. In 8th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2004), Oslo, Norway. 2004.

[Goulão 04d] Miguel Goulão & Fernando Brito Abreu. *Software Components Evaluation: an Overview*. In 5ª Conferência da APSI (CAPSI 2004), Lisbon. 2004.

[Goulão 05a] Miguel Goulão & Fernando Brito Abreu. *Composition Assessment Metrics for CBSE*. In 31st Euromicro Conference - Component-Based Software Engineering Track (Euromicro'2005), Porto, Portugal. IEEE Computer Society, 2005.

[Goulão 05b] Miguel Goulão & Fernando Brito Abreu. *Formal Definition of Metrics upon the CORBA Component Model*. In Ralf Reussner, Johannes Mayer, Judith A. Stafford, Sven Overhage, Steffen Becker & Patrick J. Schroeder, editors, First International Conference on the Quality of Software Architectures (QoSA'2005), volume 3712 of *LNCS*, Erfurt, Germany. Springer, 2005.

[Goulão 05c] Miguel Goulão & Fernando Brito Abreu. *Validação Cruzada de Métricas para Componentes*. IEEE Transactions Latin America, vol. 3, no. 1, 2005.

[Goulão 06] Miguel Goulão & Fernando Brito Abreu. *On the Influence of Practitioners' Expertise in Component-Based Software Reviews*. In 10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2006), Nantes. 2006.

[Goulão 07a] Miguel Goulão & Fernando Brito Abreu. *Modeling the Experimental Software Engineering Process*. In Ricardo Machado, editor, 6th International Conference on the Quality of Information and Communications Technology (QUATIC'2007), Lisbon, Portugal. IEEE Computer Society, 2007.

[Goulão 07b] Miguel Goulão & Fernando Brito Abreu. *An overview of metrics-based approaches to support software components reusability assessment*. In Ravi Kumar Jain B., editor, Software Quality Measurement: Concepts and Approaches, page 264. ICFAI Books, Hyderabad, 2007.

[Grassi 05] Vincenzo Grassi, Raffaela Mirandola & Antonio Sabetta. *An XML-Based Language to Support Performance and Reliability Analysis in Software Architectures*. In Ralf Reussner, Johannes Mayer, Judith A. Stafford, Sven Overhage, Steffen Becker & Patrick J. Schroeder, editors, First International Conference on the Quality of Software Architectures (QoSA'2005), volume 3712 of *LNCS*, pages 71–87, Erfurt, Germany. Springer, 2005.

[Guerreiro 01] Pedro Guerreiro. *Simple Support for Design by Contract in C++*. In TOOLS USA 2001, pages 24–34, Santa Barbara, CA, USA. IEEE Computer Society, 2001.

[Gursaran 01] Gursaran & Gurdev Roy. *On the Applicability of Weyuker Property 9 to Object-Oriented Structural Inheritance Complexity Metrics*. IEEE Transactions on Software Engineering, vol. 27, no. 4, pages 381–384, 2001.

[Hamilton 97] Graham Hamilton. *JavaBeans (version 1.01-A)*. Api specification, Sun Microsystems, August 1997.

[Heineman 01] G. T. Heineman & W. T. Councill. Component-Based Software Engineering - Putting the Pieces Together. Addison-Wesley, Boston, MA, 2001.

[Henderson-Sellers 02] Brian Henderson-Sellers, F. Stallinger & B. Lefever. *The OOSPICE Methodology Component: Creating a CBD Process Standard*. In Frank Barbier, editor, Business Component-Based Soft-

ware Engineering, The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston Hardbound, 2002.

[Hoek 03] André van der Hoek, Ebru Dincel & Nenad Medvidovic. *Using Service Utilization Metrics to Assess and Improve Product Line Architectures*. In 9th IEEE International Software Metrics Symposium (Metrics'2003), Sydney, Australia. IEEE Computer Society Press, 2003.

[Höst 00] Martin Höst, Björn Regnell & Claes Wohlin. *Using Students as Subjects - A Comparative Study of Students and Professionals in Lead-Time Impact Assessment*. Empirical Software Engineering, vol. 5, no. 3, pages 201–214, 2000.

[Inoue 05] Katsuro Inoue, Reishi Yokomori, Tetsuo Yamamoto, Makoto Matsushita & Shinji Kusumoto. *Ranking Significance of Software Components Based on Use Relations*. IEEE Transactions on Software Engineering, vol. 31, no. 3, pages 213–225, 2005.

[ISBSG 07a] ISBSG. *ISBSG Data Demographics*. Technical report, International Software Benchmarking Standards Group, January 2007.

[ISBSG 07b] ISBSG. *ISBSG Repository Data Release 10 - Field Descriptors*. Technical report, International Software Benchmarking Standards Group, January 2007.

[ISO15504 98] ISO15504. *Software Process Improvement and Capability dEtermination*, 1998.

[ISO9126 01] ISO9126. *ISO/IEC 9126: Information Technology - Software Product Evaluation - Software Quality Characteristics and Metrics*, 1995 2001.

[Jedlitschka 04] Andreas Jedlitschka & Marcus Ciolkowski. *Towards Evidence in Software Engineering*. In International Symposium on Empirical Software Engineering (ISESE'04), pages 261–270. IEEE Computer Society, 2004.

[Jedlitschka 05a] Andreas Jedlitschka & Marcus Ciolkowski. *Guidelines for Empirical Work in Software Engineering*. Technical Report IESE-Report No. 053.05/E (Version 1.0), Fraunhofer Institute for Experimental Software Engineering, August 2005.

[Jedlitschka 05b] Andreas Jedlitschka & Dietmar Pfahl. *Reporting Guidelines for Controlled Experiments in Software Engineering*. In 4th International Symposium on Empirical Software Engineering (ISESE 2005), pages 95–104, Noosa Heads, Australia. IEEE Computer Society, 2005.

[Jeusfeld 98] Manfred A. Jeusfeld, Christoph Quix & Matthias Jarke. *Design and Analysis of Quality Information for Data Warehouses*. In International Conference on Conceptual Modeling / the Entity Relationship Approach, pages 349–362, 1998.

[Jones 90] Cliff B. Jones. Systematic Software Development Using VDM. Prentice-Hall International, Hemel Hempstead (U.K.), 2nd (first in 1986) edition, 1990.

[Josuttis 99] Nicolai M. Josuttis. The C++ Standard Library: A Tutorial and Reference. Addison-Wesley Publishing Company, 1999.

[Juristo 98] Natalia Juristo & Ana M. Moreno. *An Adaptation of Experimental Design to the Empirical Validation of Software Engineering Theories*. Technical report, Nasa Goddard Space Flight Center, 1998.

[Juristo 01] Natalia Juristo & Ana M. Moreno. Basics of Software Engineering Experimentation. Kluwer Academic Publisher, 2001.

[Katz 82] Ralph Katz & Thomas J. Allen. *Investigating the Not Invented Here (NIH) Syndrome : A Look at the Performance, Tenure, and Communication Patterns of 50 R&D Project Groups*. R&D Management, vol. 12, pages 7–19, 1982.

[Kerievsky 05] Joshua Kerievsky. Refactoring to Patterns. The Addison-Wesley Signature Series. Addison Wesley, 2005.

[Kernighan 88] Brian W. Kernighan & Dennis M. Ritchie. The C Programming Language. Prentice Hall, Englewood Cliffs, NJ, EUA, 2nd (1st in 1978) edition, 1988.

[Kitchenham 95] Barbara Ann Kitchenham, Shari Lawrence Pfleeger & Norman Fenton. *Towards a Framework for Software Measurement Validation*. IEEE Transactions on Software Engineering, vol. 21, no. 12, pages 929–944, 1995.

[Kitchenham 96a] Barbara Ann Kitchenham. *Evaluating Software Engineering methods and tool - Part 1: The evaluation context and evaluation methods*. ACM SIGSOFT Software Engineering Notes, vol. 21, no. 1, pages 11–14, 1996.

[Kitchenham 96b] Barbara Ann Kitchenham. *Evaluating Software Engineering Methods and Tool - Part 3: Selecting an appropriate evaluation method – practical issues*. ACM SIGSOFT Software Engineering Notes, vol. 21, no. 4, pages 9–12, 1996.

[Kitchenham 01] Barbara Ann Kitchenham & Robert T. Hughes. *Modeling Software Measurement Data*. IEEE Transactions on Software Engineering, vol. 27, no. 9, pages 788–804, 2001.

[Kitchenham 02] Barbara Ann Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam & Jarrett Rosenberg. *Preliminary Guidelines for Empirical Research in Software Engineering*. IEEE Transactions on Software Engineering, vol. 28, no. 8, pages 721–734, 2002.

[Kitchenham 04] Barbara Ann Kitchenham, Tore Dybå & Magne Jørgensen. *Evidence-based Software Engineering*. In 26th International Conference on Software Engineering (ICSE 2004), pages 273–281, Edinburgh, Scotland. IEEE Computer Society Press, 2004.

[Kitchenham 08] Barbara Ann Kitchenham, Hiyam Al-Khilidar, Muhammed Ali Babar, Mike Berry, Karl Cox, Jacky Keung, Felicia Kurniawati, Mark Staples, He Zhang & Limimg Zhu. *Evaluating guidelines for reporting empirical software engineering studies*. Empirical Software Engineering, vol. 13, no. 1, pages 97–121, 2008.

[Knight 93] John C. Knight & E. Ann Myers. *An Improved Inspection Technique*. Communications of the ACM, vol. 36, no. 11, pages 51–61, 1993.

[Koning 94] Ross Koning. *The Scientific Method*. Plant physiology information website, `http://plantphys.info/Plants_Human/scimeth.html`. accessed on 2006/07/04., Eastern Conneticut State University, 1994.

[Kramer 98] R. Kramer. *iContract - The Java Design by Contract Tool*. In TOOLS'98 USA, pages 295–307, Santa Barbara, CA, EUA. 1998.

[Krueger 92] Charles W. Krueger. *Software Reuse*. ACM Computing Surveys, vol. 24, no. 2, pages 131–183, 1992.

[Kruger 99] Justin Kruger & David Dunning. *Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments*. Journal of Personality and Social Psycology, vol. 77, no. 6, pages 1121–1134, 1999.

[Kruskal 52] William H. Kruskal & W. Wallis Allen. *Use of ranks in one-criterion variance analysis*. Journal of the American Statistical Association, vol. 47, no. 260, pages 583–621, 1952.

[Laitenberger 00] Oliver Laitenberger & Jean-Marc DeBaud. *An Encompassing Life-Cycle Centric Survey on Software Inspection*. Journal for Systems and Software, vol. 50, no. 1, pages 5–31, 2000.

[Laitenberger 02] Oliver Laitenberger. *A Survey on Software Inspection Technologies*. In S. K. Chang, editor, Handbook on Software Engineering and Knowledge Engineering, volume 2, pages 517–555. World Scientific Publishing Co., 2002.

[Larsson 04] Magnus Larsson. *Predicting Quality Attributes in Component-based Software Systems*. Phd, Mälardalen University, 2004.

[Lau 05a] Kung-Kiu Lau & Zheng Wang. *A Survey of Software Component Models*. Technical Report CSPP-30, University of Manchester, School of Computer Science, April 2005.

[Lau 05b] Kung-Kiu Lau & Zheng Wang. *A Taxonomy of Software Component Models*. In 31st Euromicro Conference - Component-Based Software Engineering Track (Euromicro 2005), Porto, Portugal. IEEE Computer Society, 2005.

[Lau 07] Kung-Kiu Lau & Zheng Wang. *Software Component Models*. IEEE Transactions on Software Engineering, vol. 33, no. 10, pages 709–724, 2007.

[Lüders 02] Frank Lüders, Kung-Kiu Lau & Shui-Ming Ho. *Specification of Software Components*. In Ivica Crnkovic & Magnus Larsson, editors, Building Reliable Component-Based Software Systems, Artech House Computing Library, pages 23–38. Artech House, Boston, 1 edition, 2002.

[Li 05] Shuyu Li, XiaoJiang Li & Jian Wu. *Components and Contracts for Embedded Software*. In 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05). IEEE Computer Society, 2005.

[Lorenz 94] Mark Lorenz & Jeff Kidd. Object-Oriented Software Metrics: A Practical Guide. Prentice Hall, Englewood Cliffs, NJ, USA, 1994.

[Lublinsky 04] Boris Lublinsky. *The Key to Superior EJB Design - Decrease network traffic in EJB implementations*. Java Developer's Journal, 2004.

[Luckham 95] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Veera, D. Brian & W. Mann. *Specification and analysis of system architecture using Rapide*. IEEE Transactions on Software Engineering, vol. 21, no. 4, pages 336–355, 1995.

[Ma 04] Haohai Ma, Weizhong Shao, Lu Zhang, Zhiyi Ma & Yanbing Jiang. *Applying OO Metrics to Assess UML Meta-models*. In 7th International Conference on Modelling Languages and Applications («UML» 2004), volume 3273 of *LNCS*, pages 12–26, Lisbon, Portugal. Springer, 2004.

[Mach 05] M. Mach, Frantisek Plásil & J. Kofron. *Behavior Protocol Verification: Fighting State Explosion*. International Journal of Computer and Information Science, vol. 6, no. 1, pages 22–30, 2005.

[Madachy 93] R. Madachy, L. Little & S. Fan. *Analysis of a successful inspection program*. In 18th Annual Nasa Software Engineering Laboratory Workshop, pages 176–198, 1993.

[Magee 95] Jeff Magee, Naranker Dulay, Susan Eisenbach & Jeff Kramer. *Specifying distributed software architectures*. In Wilhelm Schäfer & Pere Botella, editors, Fifth European Software Engineering Conference (ESEC'95), volume 989 of *LNCS*, pages 137–153, Sitges, Spain. Springer, 1995.

[Maroco 03] João Maroco. Análise Estatística - Com Utilização do SPSS. Edições Sílabo, Lisbon, 2nd edition, 2003.

[McIlroy 69] M. D. McIlroy. *Mass Produced Software Components*. In P. Naur & B. Randell, editors, Software Engineering, Report on a conference by the NATO Science Committee. NATO Scientific Affairs Division, Brussels, Belgium, pp. 138-150., volume 1, pages 138–150. NATO Science Committee, Garmisch, Germany, 1969.

[Medvidovic 96] Nenad Medvidovic, P. Oreizy, Jason E. Robbins & Richard N. Taylor. *Using object-oriented typing to support architectural design in the C2 style*. In Fourth ACM Symposium on the Foundations of Software Engineering (SIGSOFT'96). ACM Press, 1996.

[Medvidovic 02] Nenad Medvidovic, David S. Roseblum, David F. Redmiles & Jason E. Robbins. *Modeling Software Architectures in the Unified*

Modeling Language. ACM Transactions on Software Engineering and Methodology, vol. 11, no. 1, pages 2–57, 2002.

[Merle 03] Philippe Merle. *DTD for Component Assembly Descriptor defined by the CORBA Components Specification 3.0*, 2003.

[Meyer 92a] Bertrand Meyer. *Applying "Design by Contract"*. IEEE Computer, vol. 25, no. 10, pages 40–51, 1992.

[Meyer 92b] Bertrand Meyer. Eiffel: The Language. Prentice Hall, 2nd edition, 1992.

[Meyer 00] Bertrand Meyer. *Contracts for components*. Software Development Magazine, July 2000.

[Meyer 06] Bertand Meyer & Karine Arnout. *Componentization: The Visitor Example*. IEEE Computer, vol. 39, no. 7, pages 23–30, 2006.

[Microsoft 96] Microsoft. *The Component Object Model Specification*, 1996.

[Miller 56] George A. Miller. *The Magical Number Seven, Plus or Minus Two : Some limits in our Capacity for Processing Information*. The Psychological Review, vol. 63, pages 81–97, 1956.

[Miller 98] James Miller, Murray Wood, Marc Roper & Andrew Brooks. *Further Experiences with Scenarios and Checklists*. Empirical Software Engineering, vol. 3, no. 1, pages 37–64, 1998.

[Miller 00] James Miller. *Applying Meta-Analytical Procedures to Software Engineering Experiments*. Journal of Systems and Software, vol. 54, no. 11, pages 29–39, 2000.

[Milner 92] Robin Milner, Joachim Parrow & David Walker. *A Calculus of Mobile Processes, parts I and II*. Journal of Information and Computation, vol. 100, pages 1–77, 1992.

[Milner 93] Robin Milner. *The polyadic π-calculus: a tutorial*. In Logic and Algebra of Specification, pages 203–246. Springer, 1993.

[Mohagheghi 07] Parastoo Mohagheghi & Reidar Conradi. *Quality, productivity and economic benefits of software reuse: a review of industrial studies*. Empirical Software Engineering, vol. 12, pages 471–516, 2007.

[Moreno 05] Gabriel Moreno, Scott A. Hissam & Kurt Wallnau. *Statistical Models for Empirical Component Properties and Assembly-Level Property Predictions: Toward Standard Labeling*. In 5th ICSE

Workshop on Component-Based Software Engineering (CBSE 2005), Orlando, Florida. 2005.

[Moriconi 95] M. Moriconi, X. Qian & R. Riemenschneider. *Correct architecture refinement*. IEEE Transactions on Software Engineering, vol. 21, no. 4, pages 356–373, 1995.

[Morisio 99] Maurisio Morisio. *Measurement processes are software, too*. Journal for Systems and Software, vol. 49, pages 17–31, 1999.

[Moser 97] Simon Moser & Vojislav B. Misic. *Measuring class coupling and cohesion: a formal metamodel approach*. In Asia-Pacific Software Engineering Conference (APSEC 1997) and International Computer Science Conference (ICSC 1997), pages 31–40, 1997.

[Narasimhan 04] V. Lakshmi Narasimhan & Baju Hendradjaya. *A New Suite of Metrics for the Integration of Software Components*. In Henry Detmold, Katrina Falkner & David S. Munro, editors, The First International Workshop on Object Systems and Software Architectures (WOSSA'2004), South Australia, Australia. The University of Adelaide, 2004.

[Nierstrasz 92] Oscar Nierstrasz, Simon Gibbs & Dennis Tsichritzis. *Component-Oriented Software Development*. Communications of the ACM, vol. 35, no. 9, pages 160–165, 1992.

[Nierstrasz 02] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, Andrew P. Black, Peter Müller, Christian Zeidler, Thomas Genssler & Reiner van der Born. *A Component Model for Field Devices*. In Judith Bishop, editor, IFIP/ACM Working Conference on Component Deployment, volume LNCS 2370, pages 200–209, Berlin, Germany. Springer, 2002.

[Ning 96] Jim Q. Ning. *A Component-Based Software Development Model*. In 20th International Computer Software and Applications Conference (COMPSAC 1996), pages 389–394, 1996.

[OMG 02a] OMG. *CORBA Components - Version 3.0*. Specification formal/02-06-65, Object Management Group Inc., June 2002.

[OMG 02b] OMG. *Meta Object Facility (MOF) Specification (Version 1.4)*. Technical report, Object Management Group, April 2002.

[OMG 03a] OMG. *Common Warehouse Metamodel (CWM) Specification (Version 1.1, Volume 1)*. Technical Report formal/03-03-02, Object Management Group Inc., March 2003.

[OMG 03b] OMG. *UML 2.0 OCL Final Adopted specification*. Technical Report ptc/03-10-14, Object Management Group Inc., October 2003.

[OMG 04] OMG. *Meta Object Facility (MOF) 2.0 Core Specification*. Technical Report ptc/04-10-15, Object Management Group Inc., 2004.

[OMG 05a] OMG. *UML Profile for Schedulability, Performance, and Time - version 1.1*. Technical Report formal/05-01-02, Object Management Group Inc., January 2005.

[OMG 05b] OMG. *Unified Modeling Language: Superstructure - version 2.0*. Technical Report formal/05-07-04, Object Management Group Inc., August 2005.

[OMG 06a] OMG. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms - OMG Available Specification, version 1.0*. OMG Available Specification formal/06-05-02, Object Management Group Inc., May 2006.

[OMG 06b] OMG. *Unified Modeling Language: Infrastructure - version 2.0*. Technical Report formal/05-07-05, Object Management Group Inc., March 2006.

[OMG 07] OMG. *Unified Modeling Language: Superstructure - version 2.1.1*. Technical Report formal/2007-02-05, Object Management Group Inc., February 2007.

[Ommering 00] Rob van Ommering, Frank van der Linden, Jeff Kramer & Jeff Magee. *The Koala Component Model for Consumer Electronics Software*. IEEE Computer, vol. 33, no. 3, pages 78–85, 2000.

[Ommering 04] Rob van Ommering. *Builiding Product Populations with Software Components*. Phd, Rijksuniveriteit Groningen, 2004.

[Pai 04] Madhukar Pai, Michael McCulloch, Jennifer D. Gorman, Nikita Pai, Wayne Enanoria, Gail Kennedy, Prathap Tharian & Jr Colford John M. *Systematic reviews and meta-analyses: An illustrated step-by-step guide*. National Medical Journal of India, vol. 17, no. 2, pages 86–95, 2004.

[Parnas 85] David Lorge Parnas & D. M. Weiss. *Active Design Reviews: Principles and Practices*. In 8th International Conference on Software Engineering (ICSE'85), pages 215–222, 1985.

[Parnas 87] David Lorge Parnas & D. M. Weiss. *Active Design Reviews: Principles and Practices*. Journal of Systems and Software, vol. 4, no. 7, pages 259–265, 1987.

[Plauger 91] P. J. Plauger. The Standard C Library. Prentice Hall, 1991.

[Plösh 04] Reinhold Plösh. Contracts, Scenarios and Prototypes: An Integrated Approach to High Quality Software. Springer, Berlin, 2004.

[Plásil 98] Frantisek Plásil, Dusan Bálek & Radovan Janecek. *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*. In Fourth International Conference on Configurable Distributed Systems (ICCDS 1998), pages 43–51, Annapolis, MA, USA. IEEE Computer Society, 1998.

[Polák 05] Matej Polák. *UML 2.0 Components*. Msc., Faculty of Mathematics and Physics, Charles University, 2005.

[Porter 95] Adam A. Porter, Lawrence G. Votta & Victor R. Basili. *Comparing Detection Methods for Software Requirement Inspections: a Replicated Experiment*. IEEE Transactions on Software Engineering, vol. 21, no. 6, pages 563–575, 1995.

[Porter 97a] Adam Porter & Philip Johnson. *Assessing Software Review Meetings: Results of a Comparative Analysis of Two Experimental Studies*. IEEE Transactions on Software Engineering, vol. 23, no. 3, pages 129–145, 1997.

[Porter 97b] Adam Porter & Lawrence Votta. *What Makes Inspections Work?* IEEE Software, vol. 14, no. 6, pages 99–102, 1997.

[Porter 97c] Adam A. Porter, Harvey Siy, C. A. Toman & Lawrence G. Votta. *An experiment to assess the cost-benefits of code inspections in large scale software development*. IEEE Transactions on Software Engineering, vol. 23, no. 6, pages 329–346, 1997.

[Porter 98] Adam Porter, Harvey Siy, Audris Mockus & Lawrence Votta. *Understanding the sources of variation in software inspections*. ACM Transactions on Software Engineering and Methodology, vol. 7, no. 1, pages 41–79, 1998.

[Ramesh 04] V. Ramesh, Robert L. Glass & Iris Vessey. *Research in computer science: an empirical study*. Information and Software Technology, vol. 70, pages 165–176, 2004.

[Richters 01] Mark Richters. *A UML-based Specification Environment*, 2001.

[Robbins 98] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles & David S. Rosenblum. *Integrating Architecture Description Languages with a Standard Design Method*. In International Conference on Software Engineering (ICSE98), Kyoto, Japan. 1998.

[Roh 04] Sunghwan Roh, Kyungrae Kim & Taewoong Jeon. *Architecture modeling language based on UML2.0*. In 11th Asia-Pacific Software Engineering Conference, pages 663–669, 2004.

[Royce 70] W. W. Royce. *Managing the Development of Large Software Systems*. In IEEE WESCON 1970, pages 1–9, 1970.

[Runeson 03] Per Runeson. *Using Students as Experiment Subjects - An Analysis on Graduate and Freshmen Student Data*. In 7th International Conference on Empirical Assessment in Software Engineering (EASE 2003), 2003.

[Sahraoui 01] Houari A. Sahraoui, Mounir Boukadoum & Hakim Lounis. *Building Quality Estimation Models with Fuzzy Threshold Values*. L'Objet, vol. 17, no. 4, pages 535–554, 2001.

[Sauer 00] Chris Sauer, Ross Jeffery, Lesley Land & Philip Yetton. *The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research*. IEEE Transactions on Software Engineering, vol. 26, no. 1, pages 1–14, 2000.

[Schmidt 99] Douglas C. Schmidt. *Why Software Reuse has Failed and How to Make It Work for You*. C++ Report, vol. 11, no. 1, 1999.

[Sedigh-Ali 01] Sahra Sedigh-Ali, Arif Ghafoor & Raymond A. Paul. *Software Engineering Metrics for COTS-Based Systems*. IEEE Computer, 2001.

[Seker 04] Remzi Seker, Alta van der Merwe, Paula Kotze, Murat Mehmet Tanik & R. Paul. *Assessment of Coupling and Cohesion for Component-Based Software by Using Shannon Languages*. Journal of Integrated Design & Process Science, vol. 8, no. 4, pages 33–43, 2004.

[Selic 02] Bran Selic. *On Modeling Architectural Structures with UML*. In ICSE 2002 Workshop Methods and Techniques for Software Architecture Review and Assessment, Orlando, Florida, USA. 2002.

[Sharma 06] Naveen Sharma, Padmaja Joshi & Rushikesh K. Joshi. *Applicability of Weyuker's Property 9 to Object Oriented Metrics*. IEEE Transactions on Software Engineering, vol. 32, no. 3, pages 209–211, 2006.

[Shaw 95a] Mary Shaw. *Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging*. In ACM SIGSOFT Symposium on Software Reusability, pages 3–6. ACM Press, 1995.

[Shaw 95b] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young & Gregory Zelesnik. *Abstractions for Software Architecture and Tools to Support Them*. IEEE Transactions on Software Engineering, vol. 21, no. 4, pages 314–335, 1995.

[Shaw 96] Mary Shaw. *Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does*. In 8th International Workshop on Software Specification and Design (IWSSD '96), pages 181–185. IEEE Computer Society, 1996.

[Shepperd 91] Martin Shepperd & Darrel Ince. *Algebraic Validation of Software Metrics*. In 3rd European Software Engineering Conference (ESEC'91), Milan, Italy. 1991.

[Shull 02] Forrest Shull, Victor R. Basili, Jeffrey Carver, José C. Maldonado, Guilherme Horta Travassos, Manoel Mendonça & Sandra Fabbri. *Replicating software engineering experiments: addressing the tacit knowledge problem*. In 2002 International Symposium on Empirical Software Engineering (ISESE'02), pages 7–16. IEEE Computer Society, 2002.

[Shull 04] Forrest Shull, Manoel Mendonça, Victor R. Basili, Jeffrey Carver, José Carlos Maldonado, Sandra Fabbri, Guilherme Horta Travassos & Maria Cristina Ferreira. *Knowledge-Sharing Issues in Experimental Software Engineering*. Empirical Software Engineering, vol. 9, no. 1-2, pages 111–137, 2004.

[Simão 03] Régis P. S. Simão & Arnaldo D. Belchior. *Quality Characteristics for Software Components: Hierarchy and Quality Guides*. In Alejandra Cechich, Mario Piattini & Antonio Vallecillo, editors, Component-Based Software Quality: Methods and Techniques, LNCS 2693, pages 184–206. Springer, 2003.

[Singer 99] Janice Singer. *Using the American Psychological Association (APA) Style Guidelines to Report Experimental Results*. In Work-

shop on Empirical Studies in Software Engineering (WSESE 1999), pages 71–75, Oxford, England. 1999.

[Siy 96] Harvey Siy. *Identifying the Mechanisms to Improve Code Inspection Costs and Benefits*. Phd, University of Maryland, 1996.

[Sjøberg 05] Dag I. K. Sjøberg, Jo E. Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, Nils-Kristian Liborg & Anette Rekdal. *A survey of controlled experiments in software engineering*. IEEE Transactions on Software Engineering, vol. 31, no. 9, pages 733–753, 2005.

[Sommerville 06] Ian Sommerville. Software Engineering. Addison Wesley, 8 edition, 2006.

[Stafford 01a] Judith A. Stafford & Kurt Wallnau. *Component Composition and Integration*. In Ivica Crnkovic & Magnus Larsson, editors, Building Reliable Component-Based Software Systems, pages 179–191. Artech House, Boston, London, 2001.

[Stafford 01b] Judith A. Stafford & Kurt Wallnau. *Is Third Party Certification Necessary?* In 4th ICSE Workshop on Component-Based Software Engineering (CBSE 2001), Toronto, Canada. 2001.

[Stewart 95] W. J. Stewart. Introduction to the Numerical Solution of Markov Chains. Princeton University Press, 1995.

[Szyperski 02] Clemens Szyperski, Dominic Gruntz & Stephan Murer. Component Software: Beyond Object-Oriented Programming. Component Software Series. ACM Press - Addison Wesley, New York, 2nd edition, 2002.

[Tichy 95] Walter Tichy, Paul Lukowicz, Lutz Prechelt & Ernst A. Heinz. *Experimental Evaluation in Computer Science: A Quantitative Study*. Journal for Systems and Software, vol. 28, no. 1, pages 9–18, 1995.

[Tichy 98] Walter Tichy. *Should Computer Scientists Experiment More?* IEEE Computer, pages 32–40, 1998.

[Trochim 06] William M. Trochim. *The Research Methods Knowledge Base, 2nd Edition.* `http://trochim.human.cornell.edu/kb/index.htm`, (accessed on 2006/08/10) 2006.

[Vogelson 01] Cullen T. Vogelson. *Seeking the perfect protocol*. Modern Drug Discovery: from concept to development, American Chemical Society Press, vol. 4, no. 1, 2001.

[W3C 04] W3C. *Web Services Architecture*. Technical report, W3C, February 2004.

[Wallnau 01] Kurt Wallnau & Judith A. Stafford. *Ensembles: abstractions for a new class of design problem*. In 27th Euromicro Conference (Euromicro 2001), pages 48–55, Warsaw, Poland. IEEE Computer Society, 2001.

[Wallnau 02] Kurt Wallnau & Judith A. Stafford. *Dispelling the Myth of Component Evaluation*. In Ivica Crnkovic & Magnus Larsson, editors, Building Reliable Component-Based Software Systems, pages 157–177. Artech House, Boston, London, 2002.

[Wallnau 03] Kurt Wallnau. *Volume III: A Technology for Predictable Assembly from Certifiable Components*. Technical Report CMU/SEI-2003-TR-009, Carnegie Mellon, Software Engineering Institute, April 2003.

[Wang 01] Nanbor Wang, Douglas C. Schmidt & Carlos O'Ryan. *Overview of the CORBA Component Model*. In George T. Heineman & William T. Councill, editors, Component-Based Software Engineering: Putting the Pieces Together, pages 557–571. Addison-Wesley Publishing Company, 1st edition, 2001.

[Wang 04] Nanbor Wang, Christopher Gill, Venkita Subramanian & Douglas C. Schmidt. *Configuring Real-time Aspects in Component Middleware*. In International Symposium on Distributed Objects and Applications, pages 1520–1537, Agia Napa, Cyprus. 2004.

[Washizaki 03] Hironori Washizaki, Hirokazu Yamamoto & Yoshiaki Fukazawa. *A Metrics Suite for Measuring Reusability of Software Components*. In 9th IEEE International Software Metrics Symposium (METRICS 2003), Sydney, Australia. IEEE Computer Society, 2003.

[Weller 93] Edward F. Weller. *Lessons from Three Years of Inspection Data*. IEEE Software, vol. 10, no. 5, pages 38–45, 1993.

[Weyuker 88] Elaine J. Weyuker. *Evaluating Software Complexity Measures*. IEEE Transactions on Software Engineering, vol. 14, no. 9, pages 1357–1365, 1988.

[Winter 02] Michael Winter, Thomas Gensler, Alexander Christoph, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Peter Müller, Chris Stich & Bastiaan Schönhage. *Components for Embedded Software - The PECOS Approach*. In Second ECOOP International Workshop on Composition Languages (WCL 2002), Málaga, Spain. 2002.

[Wohlin 99] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell & A. Wesslén. Experimentation in Software Engineering: An Introduction, volume 6. Kluwer Academic Publishers, Boston, EUA, 1999.

[Wohlin 02] Claes Wohlin, Aybuke Aurun, Håkan Petersson, Forrest Shull & Marcus Ciolkowski. *Software Inspection Benchmarking - A Qualitative and Quantitative Comparative Opportunity*. In Eighth IEEE Symposium on Software Metrics (METRICS'2002), pages 118–127. IEEE Computer Society, 2002.

[Wolfs 96] Frank Wolfs. *Introduction to the Scientific Method*. `http://teacher.pas.rochester.edu/phy_labs/AppendixE/AppendixE.html` (accessed on 2006/07/04), University of Rochester, 1996.

[Wudka 98] Jose Wudka. *Physics 7 Notes*. `http://physics.ucr.edu/~wudka/Physics7/Notes_www/node5.html` (accessed on 2006/07/04)., University of California Riverside, 1998.

[Yohai 87] Victor J. Yohai. *High breakdown-point and high efficiency robust estimates for regression*. The Annals of Statistics, vol. 15, no. 20, pages 642–656, 1987.

[Zelkowitz 96] Marvin V. Zelkowitz & Dolores Wallace. *Experimental Models for Software Diagnosis*. Technical Report NISTIR 5889, National Institute of Standards and Technology, September 1996.

[Zelkowitz 97] Marvin V. Zelkowitz & Dolores Wallace. *Experimental Validation in Software Engineering*. Journal of Information and Software Technology, vol. 39, pages 735–743, 1997.

[Zhang 02] Lu Zhang & Dan Xie. *Comments on "On the Applicability of Weyuker Property 9 to Object-Oriented Structural Inheritance Complexity Metrics*. IEEE Transactions on Software Engineering, vol. 28, no. 5, pages 526–527, 2002.